

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H.J. Schneider

EBIDIS

STRUKTURIERTE BILDPROGRAMMIERUNG IM DIALOG

Hans - Werner Brüggmann

1977

Dissertation

BAND 10 · NUMMER 17 · ERLANGEN · DEZEMBER 1977

1978 # 46



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546228_0115

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER : H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees,
H. Niemann, D. Seitzer, H.J. Schneider

E B I D I S

STRUKTURIERTE BILDPROGRAMMIERUNG IM DIALOG

Dem Fachbereich Ingenieurwissenschaften
an der Friedrich-Alexander-Universität Erlangen-Nürnberg
zur Erlangung des Grades
D o k t o r - I n g e n i e u r

vorgelegt

von

Hans-Werner Brüggmann

Dissertation

BAND 10 · NUMMER 17 · ERLANGEN · DEZEMBER 1977



Als Dissertation genehmigt vom
Fachbereich Ingenieurwissenschaften
der Universität Erlangen-Nürnberg

Tag der Einreichung:	2. 11. 1977
Tag der Promotion:	9. 12. 1977
Dekan:	Prof. Dr. D. Seitzer
Berichterstatter:	Prof. Dr. H. J. Schneider
	Prof. Dr. F. Hofmann

meinen Eltern gewidmet

Z U S A M M E N F A S S U N G :

Das Erlanger Bild-Dialogsystem ist ein Dialogsystem für strukturierte Bildprogrammierung. Die Dialogführung geschieht durch den Benutzer in einer Eingabesprache, die durch Freiheiten in der Formulierung eine dem natürlichen Deutsch sehr ähnliche Ausdrucksweise erlaubt. Zur Beschreibung zweidimensionaler Bilder stehen Anweisungen zur Verfügung, die sich an der Bildstruktur orientieren und die Ausdrücke enthalten, welche arithmetischen Ausdrücken in höheren Programmiersprachen vergleichbar sind. Auch der Entstehungsprozeß der Bilder entspricht der Programmiermethodik bei höheren Programmiersprachen. Analog zum Programmentwurf *top-down* und *bottom-up* können Bilder schrittweise verfeinert oder aufgebaut werden. Bildmanipulationen wie Drehung oder Größenveränderung sind implizit oder durch explizite Angaben leicht möglich. In einer Bildbank werden Bilder als baumartige Datenstrukturen verwaltet. Der Dialog wird über ein graphisches Sichtgerät mit Tastatur, Lichtgriffel und Steuerknüppeln geführt.

Kapitel 1 befaßt sich mit Eigenschaften, die große Programmsysteme besitzen müssen. Entwicklungsmethoden sind natürlich eng mit den in Kapitel 1 genannten Merkmalen der entwickelten Produkte verbunden. So werden in Kapitel 2 erfolgreiche Eigenschaften der Verfahren zur Syntaxanalyse in Compilern zusammengestellt, um daraus in Kapitel 3 ein Entwurfswerkzeug für große Programmiervorhaben vorzustellen. Zugleich werden einige Eigenschaften sehr hoher Programmiersprachen diskutiert und an der Sprache CDL (Compiler Description Language) aufgezeigt. Das in Kapitel 4 beschriebene Dialogsystem EBIDIS wird in dieser Art und Weise auch implementiert. Bemerkungen zur Implementierung enthält Kapitel 5.

ABSTRACT :

EBIDIS is a dialog system for structured picture programming. The user forms the dialog by means of an input language which allows - through open formulation of phrases - a mode of expression very similar to the natural German language. For describing two-dimensional pictures, statements are available which predominantly reflect picture structure and contain expressions comparable to arithmetic expressions in higher programming languages. The process of creating pictures is equivalent to the programming methods in problem oriented programming languages. According to the program development in the top-down and bottom-up direction, pictures can be refined or built up step by step. Picture manipulations such as rotation or zooming are easily carried out in an implicit way or by explicit instructions. In a picture data base pictures are maintained as tree-structured data. The dialog is carried on by a graphical display with keyboard, light pen and joystick option.

Chapter 1 deals with qualities large software systems should have. Of course, developing methods of such systems are closely linked to the qualities of the developed products. Thus in chapter 2, successful properties of methods concerning the syntactic analysis in compilers are shown in order to apply a resulting development tool in large software projects. At the same time some properties of very high level programming languages are discussed and shown in the context of CDL (Compiler Description Language). The dialog system EBIDIS, described in chapter 4, is implemented in this way as well. Chapter 5 contains comments on implementation.

1	ANFORDERUNGEN AN GROSSE PROGRAMMSYSTEME	1
1.1	Instanzen beim Mensch-Maschine-Dialog und ihre Verbindungen	1
1.2	Zur Verbindung <i>Anwender - Fachmann</i> : Entwurf von Programmsystemen	3
1.3	Zur Verbindung <i>Fachmann - Programmsystem</i> : Prüfung und Verifikation	4
1.4	Programmsysteme von bleibendem Wert	7
2	ERRUNGENSCHAFTEN DER GRAMMATIKGESTEUERTEN SYNTAXANALYSE BEI COMPILERN	9
2.1	Formale Sprachdefinitionen	10
2.2	Analyseverfahren	11
2.2.1	Gegenüberstellung von Verfahren	11
2.2.2	Tendenzen der Tafelverfahren	13
2.2.3	Verfahren mit rekursivem Abstieg	17
2.3	Identität von Syntaxdefinition und Syntaxanalyse	19
3	CDL, EIN BEITRAG ZUR PROGRAMMIERMETHODIK	22
3.1	Problembeschreibung	22
3.2	Problemlösung	24
3.3	Der Prozeß der Lösungsfindung	26
3.3.1	Der top-down-Entwurf	26
3.3.2	Die bottom-up-Konstruktion	27
3.3.3	Synthese beider Methoden	28
3.3.4	Lösungsprozeß bei Verwendung von CDL	30
3.4	Eigenschaften von CDL-Programmen	31

	Seite
4	BILDPROGRAMMIERUNG IM DIALOG 36
4.1	Dialogischer Einsatz von Rechenanlagen 36
4.2	Allgemeine Eigenschaften des Dialogsystems 40
	EBIDIS 40
4.3	Bildbearbeitung mit EBIDIS 44
4.3.1	Tätigkeiten bei der Verwaltung von Objekten 44
4.3.1.1	Namensliste 45
4.3.1.2	Bildbank 48
4.3.1.3	Textspeicher 50
4.3.1.4	Auftragskeller 54
4.3.1.5	Dialogverwaltung 55
4.3.2	Die Dialogsprache 57
4.3.2.1	Formale Betrachtung 57
4.3.2.2	Eigenschaften realer Dialogsprachen 60
4.3.2.2.1	Ein- und Ausgabesprache 60
4.3.2.2.2	Der Einfluß des Dialogzustandes 61
4.3.2.2.3	Darstellung von Fehlersituationen 63
4.3.2.3	Die Sprache bei EBIDIS 64
4.3.2.3.1	Kommandos 65
4.3.2.3.2	Bildsprache 69
4.3.2.3.2.1	Primitive Bilder 69
4.3.2.3.2.2	Verknüpfende Operatoren 70
4.3.2.3.2.3	Bildmanipulationen 72
4.3.2.3.3	Rahmensprache 73
4.3.2.3.4	Nichtsprachliche Verständigung 75
5	BEMERKUNGEN ZUR IMPLEMENTIERUNG 78
5.1	Das EHBIS-Programmiersystem 78
5.2	Das Dialogsystem EBIDIS 79
6	DANK 80
7	LITERATURVERZEICHNIS 81

1 ANFORDERUNGEN AN GROSSE PROGRAMMSYSTEME

1.1 Instanzen beim Mensch-Maschine-Dialog und ihre Verbindungen

Wenn in der Anfangszeit der Fertigung optischer Geräte etwa ein Fernrohr in Auftrag gegeben wurde, so wußte man erst nach dem Entwurf und Bau, ob man die gewünschten Merkmale erreicht hatte. Die genaue Vergrößerung beispielsweise konnte man nicht vorausberechnen. So kam es, daß die mathematisch-physikalische Analyse optischer Geräte der handwerklichen Beherrschung nachfolgte.

Das aber ist eine grundsätzliche Beobachtung bei den Ingenieurwissenschaften. Das griechische Wort für *Techniker* bezeichnet einen Menschen, der eine ihm gestellte Aufgabe bewältigt, der mit künstlerischem und handwerklichem Geschick Methoden ersinnt und Werkzeuge erfindet, um ihm übertragene Aufgaben meisterhaft auszuführen. Das Werkzeug, mit dem sich die Praktische Informatik Aufgaben der Gesellschaft widmet, sind elektronische Rechenanlagen. Die Ausstattung moderner Rechenanlagen umfaßt insbesondere große Programmsysteme, deren Entwicklung und Herstellung meistens nach dem Prinzip "trial and error" erfolgen. Solche Programmsysteme, etwa Compiler, Betriebssysteme oder Anwendersysteme, werden benötigt, konzipiert und entwickelt, und die Leistungen des realen Produktes liegen umso näher an den ursprünglichen Zielvorstellungen, je größer die Erfahrungen der Arbeitsgruppe sind. Allerdings haben sich in unserer Zeit die Erfahrungen beim Hantieren mit großen Programmsystemen schon zu kritischen Methoden für Entwurf und Herstellung verdichtet.

Es sollen im folgenden einige Merkmale großer Programmsysteme kurz beleuchtet werden. Diese Merkmale betreffen stets Anforderungen und Leistungen, die eine Instanz stellt bzw. erbringt. Und von welchen Instanzen ist hier die Rede?

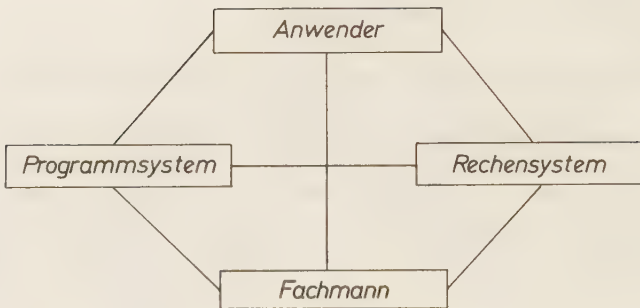


Fig. 1: Instanzen beim Mensch-Maschine-Dialog

Der Anwender, den wir uns als fachfremden Leistungsnehmer vorstellen wollen, ist noch weit davon entfernt, mit seinen Wünschen und Vorstellungen den Ton anzugeben im Bereich der elektronischen Datenverarbeitung. Trotzdem wollen wir die verschiedenen Anwendergruppen als wichtigste Instanz betrachten. Ein Anwender tritt mit einem Rechensystem in Verbindung über dessen Peripherie und mit einem Programmsystem durch ein gewisses Grundwissen über dieses System. Der direkte Austausch zwischen dem fachfremden Anwender und einem Fachmann steht für so wichtige Begriffe wie gemeinsamer Entwurf, Dokumentation und Weiterentwicklung eines DV-Systems. Er ist von entscheidender Bedeutung. Weitere in Figur 1 gezeigte Verbindungen deuten die Ausführbarkeit der programmierten Aufgaben durch ein Rechensystem an, die Programmpflege und Wartung der Anlage.

Das Diagramm zeigt also menschliche Instanzen, die überwiegend schöpferische und heuristische Leistungen erbringen, und Produkte, von denen die Masse der mechanischen Leistungen erbracht werden soll. Es zeigt auch die vielfältigen Verflechtungen aller Instanzen, die einen wirkungsvollen Mensch-Maschine-Dialog erst ermöglichen. Jede der gezeigten Verbindungen steht dabei für Bereiche, die in der Praktischen Informatik

von Bedeutung sind. Zum Beispiel umfaßt die Verbindung *Anwender - Rechensystem* Begriffe wie Entwurf und Gestaltung von Datenendgeräten, psychologische und ergonomische Aspekte, oder die Verbindung *Fachmann - Rechensystem* die Wartungsfreundlichkeit von Anlagenteilen, die Zuverlässigkeit und Prüfbarkeit von Baugruppen und Bauelementen. Da bei unserer Erörterung aber die Betonung auf *Programmsystem* liegt, sollen nicht alle Wechselwirkungen zwischen Instanzen näher untersucht werden.

1.2 Zur Verbindung Anwender - Fachmann : Entwurf von Programmsystemen

Wird ein Programmsystem nach den Anforderungen einer Benutzergruppe neu konzipiert und entwickelt, so kann man nicht von vorneherein von genauen Spezifikationen und Leistungsbeschreibungen ausgehen, sondern auch diese haben ihre Entstehungsgeschichte. Den Anfang bilden gewisse Vorstellungen über Zustände und Übergänge zugrundeliegender Datenmengen, über Lösungsmethoden und Algorithmen. Diese Vorstellungen können als Aussagen und Prädikate in natürlicher Sprache formuliert werden. Ihnen folgen Entwurfsdokumente und Problembeschreibungen sowie Problemunterteilungen. Algorithmische Sprachen gestatten es, Entwürfe und Problemstrukturen formal festzuhalten, durch höhere Programmiersprachen werden diese auf verfügbare Rechensysteme abgebildet. Dabei konkretisiert sich die Aufgabenstellung in zwei Ebenen: Es entsteht nicht nur lesbare Dokumentation, auch das Denkmodell des ursprünglichen Problems wandelt sich und paßt sich versuchsweise den vorläufigen Formalisierungen an. Hier zeigt sich ein Kriterium, nach welchem bestehende oder entstehende Programmiersprachen auch beurteilt werden müssen: Wie groß ist die suggestive Aussagekraft von logischen Einheiten in Programmtexten? Das Rekonstruieren der investierten Überlegungen muß mühelos möglich sein. Ist diese Forderung auf allen Ebenen des Entwurfs und der schrittweisen Verfeinerung

von Algorithmen und Daten erfüllt, so kommt dem Programmtext ein gewisser Dokumentationswert zu ([Wi 71], [CoGr 73]). Diese Transparenz der Aufgabenstellung im Programmtext ist von großer Bedeutung für die Lebensdauer von Programmsystemen.

1.3 Zur Verbindung Fachmann - Programmsystem: Prüfung und Verifikation

Wurde im vorigen Abschnitt in einem Programmsystem gleichsam der Kiel gelegt, so soll es nun vom Stapel laufen. Der Test- und Probetrieb gestattet es, das reale Programmverhalten mit den Zielvorstellungen beim Entwurf zu vergleichen. Diese Zielvorstellungen sind qualitativer und quantitativer Art. Qualitative Tests untersuchen, ob die gewünschten Leistungen erbracht und ob sie in der geforderten Art erbracht werden. Der Abschnitt 2.3 und das ganze Kapitel 3 beschäftigen sich mit dieser Fragestellung. Quantitative Untersuchungen beobachten die benötigten Rechenzeiten für Einzelleistungen und den Speicherbedarf hierbei. Beide Merkmale sind voneinander nicht unabhängig. Mangelnde Hauptspeicherkapazität kann man in manchen Betriebssystemen durch Überlagerungstechniken ausgleichen. Die scheinbare Vergrößerung des Hauptspeichers, der Gebrauch von virtuellem Speicher erfolgt aber dann auf Kosten erhöhter Rechenzeit und eines höheren Hintergrundspeicherbedarfs. Umgekehrt ist es nicht möglich, den Rechenzeitbedarf durch Hinzunahme von Hauptspeicher beliebig zu senken. Bei der Automatisierung von technischen Prozessen werden den Rechen- und Steuerungssystemen aber oft Grenzen für Reaktionszeiten zwingend vorgegeschrieben, deren Einhaltung sich erst bei abschließenden Testläufen zeigt. Bei dialogischer Nutzung von Rechensystemen ist die Einhaltung von gewissen Antwortzeiten aus psychologischen Gründen erforderlich. Insgesamt ist es denkbar und wünschenswert, bei Hilfsmitteln zur Programmentwicklung auch eine ständige Kontrolle über Rechenzeit- und Speicherbedarf der entwickelten Objekte zu haben einschließlich der noch fehlenden Komponenten. So können beim Überschreiten kritischer Größen schon während der Programmentwicklung Korrekturen vorgenommen und Sackgassen vermieden werden.

Qualitative Untersuchungen an Programmsystemen bewegen sich stets zwischen drei Extremen, den analytischen Methoden, dem totalen Testen und dem totalen Beweisen. Seit das Halteproblem für Turingmaschinen formuliert wurde, sind analytische Methoden zum allgemeinen Nachweis von Programmeigenschaften weitgehend ausgeschlossen. Das totale Testen eines Programmes umfaßt die Prüfung aller möglichen Programmläufe. Dijkstra führt es mit einem Beispiel ad absurdum: Ein Programm, das zwei Zahlen multipliziert, deren jede etwa zehn Dualstellen lang sei, ist denkbar einfach. Trotzdem sind über eine Million Programmläufe nötig, um für alle möglichen Eingabewerte die Ergebnisse zu prüfen. Dieses Vorgehen findet zwar bei der Kontrolle von fertigen Steckkarten Anwendung, wobei Prüfautomaten der Reihe nach alle Teste vornehmen, auf Programmebene scheitert es jedoch aus. Begnügt man sich aber mit einigen gezielten Testläufen, "so kann man damit nur die Anwesenheit, nicht die Abwesenheit von Fehlern nachweisen" (Dijkstra).

Beim Beweisen von Programmeigenschaften bzw. beim Nachweis der Richtigkeit eines Programmes oder Programmteiles verläßt man einmal die Allgemeinheit aller möglichen, in einer bestimmten Programmiersprache formulierbaren Programme. Konkretes Wissen über die Struktur eines individuellen Programmes geht wesentlich in den Beweis mit ein. Zum anderen wird die freie Verwendung der verfügbaren Sprachkonstruktionen eingeschränkt. Die ausschließliche Verwendung der Kontrollstrukturen Sequenz, Fallunterscheidung und Laufschleife, wie sie die strukturierte Programmierung fordert ([DDH 72], [Sn 75b]), ist einerseits ausreichend für die Programmierung aller berechenbaren Funktionen, begünstigt andererseits formale Programmbeweise ([Ho 69]). Diese Beweise finden bereits während der Programmerstellung statt. Schrittweise mit der Programmentwicklung wird die Gültigkeit von Aussagen über Programmgrößen nachgewiesen und aufrechterhalten. Der Nachteil von formalen Programmbeweisen liegt in ihrem Umfang. Solche Be-

weise sind meistens um ein Vielfaches länger als der eigentliche Programmtext. So ist es noch nicht unternommen worden, etwa ganze Compiler oder Betriebssysteme formal als korrekt nachzuweisen.

Programmverifikation durch formale Beweise erfolgt bereits während der Programmerstellung, ist also eine zusätzliche Aufgabe für den Implementator. Andererseits gehört der Umgang mit formalen Systemen mehr zum Handwerk des Mathematikers als in den Bereich der Praktischen Informatik. Da aber ein wichtiger Teil bei formalen Beweisen das Auffinden von Programmbeziehungen ist, welches als kreativer Vorgang mit dem Auffinden des eigentlichen Algorithmus eng zusammenhängt, bleibt diese Aufgabe doch letztlich dem Informatiker gestellt ([AlWe 77], [CoGr 73]). Am Beispiel der Laufschleifen in Programmen zeigt sich die Verwandtschaft beider Disziplinen einmal mehr. Es sind Beziehungen zu finden, welche die Wirkung eines Laufschleifenbereiches beinhalten, und es ist nachzuweisen, daß diese Beziehungen beim wiederholten Ausführen des Laufbereiches nicht verändert werden. Beides entspricht völlig der Beweismethode der Vollständigen Induktion und heißt dort Induktionsannahme bzw. Induktions-schluß. Hinzu kommt, daß Laufschleifen als Programmanweisungen ausführbare Objekte sind. Der Nachweis, daß die Ausführung einer Laufschleife unter allen Umständen zum regulären Ende kommt, ist nichttrivial und muß jedesmal gesondert geführt werden. Formales Beweisen von Programmeigenschaften ist die einzige Möglichkeit, völlige Sicherheit bezüglich der Korrektheit eines Programmes zu erreichen. Es ist eine Tätigkeit, die grundsätzlich von einem realen Rechensystem und der Ausführung des Prüflings unabhängig ist, die man aber wegen ihres Umfanges doch gerne einer Rechenanlage übertragen möchte.

Formale Programmverifikation setzt sich nur langsam durch. Das weithin praktizierte und kultivierte Testen fertiggestellter Systemkomponenten läßt sich vor dem Hintergrund des oben Ge-

sagten einordnen als informales Prüfen von Programmbeziehungen, deren Gültigkeit punktuell bei Ausführung festgestellt wird. Das geschieht in stapelorientierten Rechensystemen durch gezielt in das zu testende Programm eingestreute Ausgabeanweisungen. Verfügt man sogar über eine interaktive Ablaufsteuerung, so ist es möglich, fragliche Programmgrößen erst bei Laufunterbrechungen auszuwählen, ausgeben zu lassen und eventuell zu modifizieren ([KuWi 75]). In beiden Fällen orientiert sich der Benutzer an der Programmquelle. Um ihm bei unsystematischem, mehr heuristischem Vorgehen trotzdem eine gute Übersicht zu ermöglichen, ist wieder die Transparenz der Intention im Programmtext nötig und eine gut ausgewogene, hierarchische Programmstruktur.

1.4 Programmsysteme von bleibendem Wert

Entwurf, Entwicklung und Inbetriebnahme großer Programmsysteme sind eine spektakuläre Phase im größeren Rahmen der gesamten Einsatzdauer solcher Systeme. Alles spielt sich in der Öffentlichkeit zwischen Anwender und Implementator ab, Fortschritte und Erfolge sind unmittelbar feststellbar. Diese kurzfristigen Vorgänge werden abgelöst durch die Zeit des dauernden Einsatzes, des routinemäßigen Betriebs von Anlagen und Programmsystemen. Auch für diese Zeit gibt es Anforderungen, die aber nicht genau anzugeben noch zu überprüfen sind. Modellierbarkeit, Wandelbarkeit, Anpaßbarkeit und Übertragbarkeit von Programmen sind Merkmale, deren Vorhandensein oder Fehlen in der Anfangszeit nicht sehr auffallen. Werden sie aber im Ernstfall vermißt, so wachsen Unzufriedenheit und Ärger auf beiden Seiten, die Programmpflege wird teuer und ineffizient, Projekte werden eingeschränkt oder beendet. Die Lebendigkeit von Programmsystemen erweist sich als unkomplizierte Reaktionsfähigkeit auf sich wandelnde Umweltbedingungen. Es kann sein, daß sich die ursprüngliche Aufgabenstellung des Anwenders ändert nach Art oder Umfang, oder daß die benützte Rechenanlage durch Austausch oder Hinzunahme von Anlagenteilen und Rechnerperipherie rekonfiguriert wird. Es kann auch vorkommen, daß sich Betriebssysteme

oder Compiler als Umgebung sonstiger Anwendersysteme ändern. Wie schnell und einfach können dann notwendige Anpassungen oder auch nur erwünschte Änderungen durchgeführt werden? Wieviel Zeit müssen Mitarbeiter aufwenden, die an der Entwicklung des Programmsystems beteiligt waren, wieviel Zeit neu hinzugekommene?

Eine grobe Schätzung dafür findet sich in einem Artikel (*50 Prozent Aufschlag für Änderungsdienst*) der Zeitschrift "Computerwoche" (1977/Nr. 24): "Für den Änderungsdienst, der sich zwangsläufig aus äußeren Einflüssen (Fachabteilung, Gesetzgeber, Markt etc.) ergibt, sollte man zwischen 30 und 50 Prozent der insgesamt zur Verfügung stehenden Programmierkapazität einplanen."

Neben der Änderungsfreundlichkeit von Programmen ist die Übertragbarkeit insofern wichtig, als Programme, deren Zuverlässigkeit sich in längerem, störungsfreiem Betrieb erwiesen hat, einen hohen Wert darstellen. Beim Übergang auf eine andere Rechenanlage ähnlicher Struktur möchte man diese Werte in Form der alten, unveränderten Programmquellen erhalten. Einen Überblick über Techniken des Übertragens speziell von Compilern gibt Poole in [Po 74].

Den oben genannten Eigenschaften großer Programmsysteme kommt eine hohe Bedeutung zu. Diese Eigenschaften werden erreicht durch die Wahl geeigneter Programmiersprachen und geeigneter Programmiermethoden. Es fällt auf, daß die eingangs erwähnten Begriffe *Methoden* und *Werkzeuge* in der programmiertechnik einen festen Platz haben. Als Werkzeuge werden hierbei alle programmierten Dienstleistungen verstanden, die den Vorgang des Programmierens unterstützen, angefangen von der Texthaltung bis zur Verwaltung von binärer Information.

2 ERRUNGENSCHAFTEN DER GRAMMATIKGESTEUERTEN SYNTAXANALYSE BEI COMPILERN

Zu Beginn dieses Kapitels erhebt sich zunächst die Frage, welche Beziehungen zwischen Syntaxanalyse und Dialogsprachen bestehen. Auch für die Formulierung von Dialogschritten und den globalen Ablauf von Dialogen kann man Syntaxvorschriften angeben, derer sich interpretierende oder compilierende Dialogsysteme bei der Dialoganalyse und -führung bedienen. Sie werden in [KuWi 75] ausführlich dargestellt. Hier soll jedoch auch auf die Implementierung von Dialogsystemen eingegangen werden. Eine Arbeit von Encarnacao et al, ([En 77]) enthält ein Kapitel über Entwicklungsmethoden graphischer Systeme. Hier werden Programmiersprachen der Algol-Familie und Hardware-Beschreibungssprachen abgelehnt, weil sie Implementierungseinzelheiten widerspiegeln, andere werden abgelehnt, weil sie zu speziell (LISP) oder zu umfangreich sind (Algol 68). Entwicklungsmethoden sind natürlich eng mit den in Kapitel 1 genannten Merkmalen der entwickelten Produkte verbunden. So werden in Kapitel 2 erfolgreiche Eigenschaften der Verfahren zur Syntaxanalyse zusammengestellt, um daraus in Kapitel 3 eine Entwurfsmethodik für große Programmiervorhaben vorzustellen. Das in Kapitel 4 beschriebene Dialogsystem wird in dieser Art und Weise auch implementiert.

Die Syntaxanalyse von Programmiersprachen ist nur eine Teilaufgabe von Übersetzern, aber sicher die am besten erforschte. Die Technik der Compilerkonstruktion und die Theorie der formalen Sprachen haben sich gegenseitig befruchtet. Die höheren Programmiersprachen haben sich aus den Formelsprachen (z. B. ZUSE-Formelcode) entwickelt, deren berühmtester Vertreter heute FORTRAN ist. Als die Zeilenstruktur von Programmen zugunsten einer problemorientierten Programmstruktur überwunden wurde, da waren es die formalen Sprachen, die geeignete theoretische Modelle für Programmiersprachen darstellten und die mit korrespondierenden erkennenden Automaten auch die Grundlage für Algorithmen zur Syntaxanalyse lieferten.

2.1 Formale Sprachdefinitionen

Den Sprachen der Algol-Familie (ALGOL, SIMULA, BLISS, PASCAL, EULER u.a.) sind eine allgemeine Blockstruktur gemeinsam, die ausgiebige Verwendung von Prozeduren und verschiedenen Kontrollanweisungen. Das geschachtelte Auftreten der dabei gebildeten Programmeinheiten wird gut beschrieben durch kontextfreie Grammatiken, ihre Übersetzung geschieht grundsätzlich durch kellernde Algorithmen. Aber nicht alle Formvorschriften bei höheren Programmiersprachen sind von Natur aus kontextfrei beschreibbar. Die Sprachen der Algol-Familie kennen nur die explizite Vereinbarung von Programmgrößen für Daten und Algorithmen, und diese Vereinbarung muß in der Regel der ersten Verwendung solcher Größen im Programm vorausgehen. Diese Reihenfolgevorschrift wird bei der Übersetzung eines Programmes aber nicht im Verlauf der eigentlichen Syntaxanalyse geprüft, sondern bei zusätzlichen Untersuchungen, die meistens unter dem Begriff "semantische Programm-Analyse" zusammengefaßt werden. Genauso wird diese Vorschrift bei der Definition einer Programmiersprache nicht formal festgelegt wie die Syntax der Sprache durch eine kontextfreie Grammatik, sondern durch verbale Zusätze. Grundsätzlich ist es aber möglich, z. B. die Reihenfolgevorschrift auch formal auszudrücken. Das Hilfsmittel dazu sind die Attributgrammatiken von van Wijngaarden, die auch bei der Definition von Algol 68 verwendet wurden [vWijn 69]. Eine Weiterentwicklung der Attributgrammatiken sind die Affixgrammatiken von Koster [Ko 71a]. Beide Arten werden als 2-Stufen-Grammatiken bezeichnet.

Die erste wichtige Errungenschaft im Zusammenhang mit der Definition von Programmiersprachen war die formale Fixierung der Syntax. Die heute gebräuchlichen Hilfsmittel dafür sind die Backus-Naur-Form und einstufige van Wijngaarden-Grammatiken, beides Notationen für kontextfreie Satzbaugrammatiken, sowie die Darstellung durch Syntaxgraphen [JeWi 75]. Die van Wijngaarden-Schreibweise besitzt etliche Vorzüge gegenüber der Backus-Naur-Form.

2.2 Analyseverfahren

Die nächste große Errungenschaft bei der Übersetzung von Programmiersprachen waren die systematischen Verfahren, nach denen die Arbeit der Zerteileralgorithmen (engl. parser) in Compilern gesteuert wird. Der Programmzerteilung während der Syntaxanalyse bei einem Compilerlauf liegt die Lösung des von der Theorie der formalen Sprachen her bekannten Wortproblems zugrunde, erweitert um die Aufgabe, auch die syntaktische Struktur des ganzen Programmes zu erkennen und darzustellen. Dazu muß ein Zerteiler zwei Größen miteinander vergleichen: Die individuelle Programmquelle und die Menge aller syntaktisch zugelassenen Programme. Die Programmquelle verarbeitet ein Zerteiler bei allen praktisch eingesetzten Verfahren gleich durch einmaliges, sequentielles Einlesen. Aber wie wird das Wissen um die Menge aller gültigen Programme, wie wird die Sprachdefinition in einem Zerteiler statisch repräsentiert und zur Übersetzungszeit aktiviert? In dieser Frage unterscheiden sich die bekannten Verfahren der Syntaxanalyse wesentlich. Sieht man von den grundlegenden Unterscheidungsmerkmalen wie Rechenzeit- und Speicherplatzbedarf einmal ab, dann bleiben noch zwei wichtige Kriterien übrig. Das eine betrifft den Hersteller eines Compilers, das andere die Benutzer. Eine Programmiersprache ist Änderungen unterworfen. Was schon in Abschnitt 1.4 über geänderte Umweltbedingungen gesagt wurde, gilt hier auch für Compiler und ihre Zerteiler. Und wie gut kommt ein Übersetzer mit fehlerhaften Programmen zurecht, wie gut der Benutzer mit den vom Compiler erzeugten Fehlermeldungen? Diese Kriterien sind indirekt auch für die bunte Vielfalt von bestehenden Analyseverfahren mitverantwortlich.

2.2.1 Gegenüberstellung von Verfahren

Eine Unterscheidung von Analyseverfahren soll vorweggenommen werden: Es gab auch im Compilerbau ein "finsteres Mittelalter", in dem ein ganzer Compiler von Hand entworfen und codiert wur-

de, in dem die Zerteileralgorithmen mit Intuition und Geschick für eine zu übersetzende Programmiersprache programmiert wurden. Welche Sysiphusarbeit und welches Gratwandeln dabei vom jeweiligen Implementierer geleistet wurden, zeigt sich an den tafelgesteuerten Verfahren: Umfangreiche Manipulationen und Umformungen von Symbolmengen der zu übersetzenden Sprache erbringen die Information, die in den Steuertafeln des Zerteilers abgelegt sind. Diese Manipulationen wurden anfangs im Kopf geleistet und im Zerteileralgorithmus direkt fixiert. Damit war ein nachträgliches Ändern der so von Hand erstellten Compiler aufgrund von Sprachänderungen nahezu unmöglich.

In neuerer Zeit gehen die Bemühungen dahin, den Entwurf und die Entstehung eines Compilers vollständig zu automatisieren: Ein Compiler-Compiler übernimmt es, anhand der formalisierten Sprachdefinition und anhand von weiteren Angaben über die Rechenanlage, auf der in dieser Programmiersprache gerechnet werden soll, den gewünschten Compiler zu generieren. Für den Teilaspekt der Syntaxanalyse bedeutet das die Transformation der Sprachdefinition und ihr Integrieren in die Zerteileralgorithmen. Tafelgesteuerte Zerteiler sind hierbei aber nicht die einzige Möglichkeit. Auch in modernen Compilern kann die Menge der syntaktisch richtigen Programme ausschließlich in den Zerteileralgorithmen repräsentiert sein. Beispiele dafür sind die Compiler der Programmiersprache PASCAL ([JeWi 75]) und der Compilerbeschreibungssprache CDL ([Ko 71b]). Die Änderungsfreundlichkeit eines solchen Compilers ist dann trotzdem gegeben, wenn die Zerteileralgorithmen nicht von Hand geschrieben, sondern von einem Compiler-Compiler erzeugt wurden. Bei Tafelverfahren werden im wesentlichen einfach die Tafeln ausgetauscht. Besonders günstig ist das in dem Fall, daß aus didaktischen Gründen der erlaubte Sprachumfang einer Programmiersprache zunächst eingeschränkt und dann stufenweise wieder erweitert wird.

Anstelle der Arbeitsweise von Zerteilern - tafeigesteuert oder nicht - spricht man auch von Analysestrategien. Die Analyseverfahren, die nach der Strategie *top-down* arbeiten, kommen ohne Steuertafeln aus. Statt des ständigen Lesens in einer Tafel finden bei den Algorithmen integrierte Symbolvergleiche statt. Die Verfahren der Analysestrategie *bottom-up* arbeiten mit Tafeln, die je nach Verfahren Übergangsmatrix, Vorrangmatrix oder Aktionentafel heißen. Eine ausführliche Darstellung der Analysestrategien und wichtigsten Verfahren findet man bei [Sn 75a], [Gr 74] und [Hor 74a]. Analyseverfahren, deren Algorithmen oder Steuertafeln aus einer formalen Syntaxdefinition automatisch erzeugt wurden, bezeichnet man als syntax- oder grammatikgesteuert.

2.2.2 Tendenzen der Tafelverfahren

Mit dem wichtigsten Zerteilungsverfahren der top-down-Strategie, der Methode des rekursiven Abstiegs, wird sich der Abschnitt 2.2.3 beschäftigen. Zuvor aber sollen einige Entwicklungen bei tabellengesteuerten Verfahren kurz erwähnt werden. Dazu werden die sequentielle Formelübersetzung, das Vorrangverfahren und die LR(1)-Verfahren miteinander verglichen. Sie arbeiten deterministisch wie alle Verfahren, die im Compilerbau eingesetzt werden. Ausführliche Darstellungen der genannten Verfahren findet man in [Sn 75a], [Hor 74a] und [Br 75].

Die Steuertafel bei der sequentiellen Formelübersetzung wurde von Hand erstellt. Die Tafeleinträge beim Vorrangverfahren, einer Weiterentwicklung der sequentiellen Formelübersetzung, konnten auch maschinell aus der formal gegebenen Syntax errechnet werden. Die Tafel heißt hier *Vorrangmatrix*. Bei den LR(1)-Verfahren ist der Aufwand für die Konstruktion der Steuertafel so groß, daß nur noch die mechanische Erstellung möglich ist. In dem Maße, in welchem die Berechnung der Steuertafel automatisiert wurde, ging übrigens die Lesbarkeit der fertigen Tafel zurück. Beim Vorrangverfahren ist es dem Betrachter noch

leicht möglich, an den Einträgen in der Steuertafel die Verhältnisse in der verarbeiteten Grammatik zu erkennen, bei den LR(1)-Verfahren dagegen nur noch sehr schwer. Die Überlegenheit der LR(1)-Verfahren liegt einmal in der Übersetzung fehlerhafter Programme: Die LR(1)-Steuertafel befähigt einen Compiler, bei der Programmanalyse einen eventuellen Syntaxfehler zum frühestmöglichen Zeitpunkt zu erkennen und zu lokalisieren. Bei den Vorrangverfahren kann es geschehen, daß die Stelle einer fehlerhaften Programmfortsetzung überlesen wird, und daß der Compiler erst nach einigen weiteren Verarbeitungsschritten einen Fehler erkennt, den er dann nicht mehr analysieren kann, was für den Benutzer schon hilfreich wäre. Der andere Vorzug der LR(k)-Verfahren liegt in ihrer Mächtigkeit: Die Klasse der kontextfreien Grammatiken, deren Wortmengen mit LR(k)-Verfahren übersetzbar sind, ist wahrscheinlich die größte Klasse aller derer, die von aktuellen Verfahren übersetzt werden.

Die LR(1)-Syntaxanalyse hat noch einen großen Vorzug. Da eine Programmstelle mit einem Syntaxfehler keine versehentliche Bearbeitung erfährt, kann der Compiler versuchen, hinter dem als fehlerhaft erkannten Symbol mit der Übersetzung fortzufahren. Einerseits darf nicht zuviel Programmtext an der Fehlerstelle weggeworfen werden, andererseits muß das "gesunde Gewebe" im Kontext des Programmes erreicht werden. Die Stelle, an der weiterübersetzt wird, heißt *Aufsetzpunkt*. Sie muß so gewählt werden, daß der Compiler sich stabilisiert und nicht sofort den ersten Folgefehler meldet. Fehlertoleranz des Compilers (engl. error recovery) ist eine große Hilfe für den Benutzer, wenn die "Übersetzungslücke" keine Flut von Folgefehlern auslöst. Einzelne Folgefehler kann man dann durchaus von unabhängigen weiteren Programmfehlern unterscheiden, und die Anzahl von Übersetzungsversuchen wird drastisch gesenkt. Um die Zahl der Folgefehler einzuschränken, versuchen manche Compiler auch, den Programmfehler zu korrigieren (engl. error repair). Allerdings kann ein Compiler an Fehlerstellen die

eigentliche Intention des Programmierers nicht erraten. Horning schreibt in der Arbeit [Hör 74b], ein Compiler kann nur "korrigieren und hoffen". Ein Verfahren zur Fehlerkorrektur, das sich sehr gut in die LR(k)-Syntaxanalyse einfügt, wird in [Roe 76] beschrieben.

Der Aufwand beim Erstellen der Steuertafeln wächst in dem Maße, in welchem der Umfang an Grammatikuntersuchungen zunimmt. Sie sind für die Verfahren erforderlich und können automatisiert werden. Für ein Verfahren aus der LR(1)-Familie ist in [Br 75] ein Algorithmus dafür angegeben (*SLR(1) parsing table constructor*). Er ist in CDL beschrieben und wurde für weitere Untersuchungen schon eingesetzt. Der große Aufwand für die Berechnung von Steuertafeln ist indes kein Nachteil, denn er umfaßt auch eine kritische Behandlung der Sprachdefinition. Treten beim Erstellen einer Vorrangmatrix oder LR(1)-Tafel Mehrfachbelegungen für einzelne Matrixelemente auf, so sind Eigentümlichkeiten der Grammatik die Ursache. Die Grammatik ist nun nicht einfach "falsch", sondern das jeweilige Verfahren hat eben eine Schwachstelle. Darum klassifiziert man auch kontextfreie Grammatiken nach den Verfahren, mit denen ihre Wortmengen übersetzt werden können, und bezeichnet sie z.B. als vom Typ LL(1) oder LR(1). Die LR(k)-Verfahren haben vergleichsweise die wenigsten Schwachstellen. Eine davon ist durchaus erwünscht: Mehrdeutigkeiten in der verarbeiteten Grammatik führen immer zu Konflikten beim Aufbau der LR(1)-Tafel. Das mag verwundern, weil dieses Problem bei kontextfreien Grammatiken doch als unentscheidbar bekannt ist. Es wird auch nicht gelöst, denn der in [Br 75] verwendete Algorithmus meldet schon Konflikte, wenn nur der Verdacht auf Mehrdeutigkeiten besteht. Das soll an folgendem Beispiel gezeigt werden.

Maurer konstruiert in [Mau 69] zum Beweis der Unentscheidbarkeit des Mehrdeutigkeitsproblems zu jedem Postschen Korrespondenzproblem eine kontextfreie Grammatik und führt deren Mehrdeutigkeitsproblem auf das Korrespondenzproblem zurück.

Wir verwenden also ein Korrespondenzproblem ($X = \{A, AAB\}$, $Y = \{AA, A\}$), das sicher nicht lösbar ist, und unterwerfen die daraus entstandene Grammatik ([Mau 69], Seite 232) dem SLR(1)-Test. Das Ergebnis lautet dann folgendermaßen:

EINGEGEBENE KONTEXTFREIE GRAMMATIK :

POSTSCHES KORRESPONDENZPROBLEM :

1 XAXAABX YAAYAY, NICHT LOESBAR.

XAXAABX YAAYAY :

1 S1;

2 S2.

S1 :

1 B, A, C, A;

2 B, A, A, C, A, A, B;

3 B, A, S1, A;

4 B, A, A, S1, A, A, B.

S2 :

1 B, A, C, A, A;

2 B, A, A, C, A;

3 B, A, S2, A, A;

4 B, A, A, S2, A.

KONFLIKTE BEIM AUFBAU DER AKTIONEN TAFEL :

ZUSTAND	EINGABESYMBOL	EINGETRAGEN	UNBERUECKSICHTIGT
15	A	REDUZIEREN MIT S1/1	KELLERN MIT 20
19	A	REDUZIEREN MIT S2/2	KELLERN MIT 22

SONST KEINE KONFLIKTE

Wegen der Einzelheiten in diesen Ausgaben sei auf [Br 75] verwiesen. Obwohl die untersuchte Grammatik nicht mehrdeutig ist und das angegebene Korrespondenzproblem nicht lösbar, bewirkt seine äußere Form doch obige Konflikte. Meistens aber stecken hinter LR(1)-Konflikten doch echte Mehrdeutigkeiten,

und man kann den LR(1)-Test zum "Säubern" von Grammatiken verwenden.

Etwa zwei Drittel des rund 800 Zeilen langen CDL-Programmes für den SLR(1)-Constructor beschäftigen sich mit Untersuchungen der Grammatik und mit umfangreichen Symbolmanipulationen. Sie sind Voraussetzung für das Berechnen der SLR(1)-Tafel. Sie sind auch Grundlage für den LL(1)-Test, der im folgenden Abschnitt besprochen wird. So, wie die Syntaxanalyse einzelner Programme eine der verschiedenen Aufgaben eines Compilers ist, so gehören die Untersuchung einer Sprachdefinition und das Berechnen von Steuertafeln zu den Arbeiten eines Compiler-Compilers.

2.2.3 Verfahren mit rekursivem Abstieg

Die in Abschnitt 2.2.2 erwähnten Eigenschaften und Entwicklungen sollen nun vor dem Hintergrund der Verfahren mit rekursivem Abstieg betrachtet werden. Es sind tafelfreie Verfahren, die auch mit einem Keller arbeiten. Der nimmt aber keine Symbole auf wie bei den Tafelverfahren, sondern Prozedurdaten. Diese Prozeduren entsprechen eindeutig den Zwischensymbolen einer kontextfreien Grammatik und rufen sich gegenseitig auf, wie sich die Grammatiksymbole gegenseitig in einer Satzform erzeugen. Die Methode des rekursiven Abstiegs geht auf Lucas zurück ([Lu 61]), Knuth hat sie als *parsing machine* formalisiert ([Kn 71]). Sie ist in [Sn 75a] dargestellt. Eine allgemeine Form hat sie in den LL(k)-Verfahren gefunden, von denen das LL(1)-Verfahren häufig eingesetzt wird. Eine gründliche Darstellung des LL(1) Verfahrens gibt [Gr 74].

Das LL(1)-Analyseverfahren ist nicht so mächtig wie das LR(1)-Verfahren in seinen Versionen. Grammatiken mit linksrekursiven Symbolen und leeren Alternativen im Produktionssystem sind wohl vom Typ LR(1), bereiten aber einem LL(1)-Zerteiler Schwierigkeiten. Diese Eigenschaften einer Grammatik können zwar durch geeignete Transformationen beseitigt werden, das geschieht aber nicht beim Aufbau eines LL(1)-Zerteilers. Das Analyseverfahren einer Satzform entspricht ganz

stark dem Ableitungsprozeß derselben Satzform. Darum ist auch der direkte Übergang vom Produktionensystem der Grammatik zu den Zerteileralgorithmen möglich. Eine ausführliche Untersuchung der Sprachdefinition findet hierbei nicht statt, da auch keine Steuertafel aufgebaut werden muß. Unterbleibt die Sprachprüfung ganz, so ergeben sich eventuell Fehlleistungen des erzeugten Zerteilers. Die Eignung einer Grammatik für LL(1)-Syntaxanalyse muß also grundsätzlich unabhängig von der Erzeugung der Zerteileralgorithmen nachgewiesen werden. Ein entsprechender Test kann aber auch parallel zur Übertragung der Grammatik stattfinden. Der Aufwand für das Erstellen des Zerteilers ist geringer als der für das Berechnen von Steuertafeln. Es kommt aber der unabhängige Grammatiktest hinzu. Er kann auf der Grundlage des im vorigen Abschnitt beschriebenen SLR(1)-Constructors durchgeführt werden.

Mit den LR(1)-Verfahren hat das LL(1)-Verfahren die wichtige Eigenschaft gemeinsam, daß fehlerhafte Programmfortsetzungen zum frühestmöglichen Zeitpunkt erkannt und lokalisiert werden. Darum können auch Compiler mit einer LL(1)-Syntaxanalyse Programmfehler überwinden und fehlerhafte Programme nutzbringend weiterübersetzen. Beispiele dafür sind die CDL-Übersetzer ([Ko 71b]) und die PASCAL-Compiler (z. B. in [Br 77]). Die Berechnung von Aufsetzpunkten, die ja von der jeweils vorausgegangenen Teilanalyse abhängen, muß auch wie der Grammatiktest unabhängig von der Erzeugung der Zerteileralgorithmen durchgeführt werden.

2.3 Identität von Syntaxdefinition und Syntaxanalyse

In Abschnitt 2.2.3 klang bereits die starke Entsprechung an zwischen dem Analysevorgang derselben Satzform einer kontextfreien Grammatik, wenn die Analyse durch rekursiven Abstieg erfolgt. Durch die Formalisierung von Syntaxdefinitionen und das Entstehen systematischer Analyseverfahren wurde die Kluft zwischen dem Sprachentwurf und der Compilerimplementierung verringert. Mißverständnisse sind aber immer noch möglich. Kann man nun diese beiden Tätigkeiten, konstruktive Prozesse mit globalen Aspekten und lokaler Feinarbeit, nicht durch einen einzigen Entwurf vollbringen? In der Technik des Compilerbaues ist das gelungen. Es ist eine enorme Errungenschaft. Ähnlich bedeutungsvolle Entwicklungen konnten überall da beobachtet werden, wo formale, konsistente Systeme in ihrer eigenen Notation zu realen Rechnersprachen wurden. Nur einige Beispiele dafür sind die Produktionssprache von Floyd und Evans (s. [Sn 75a]), die sich aus Automatentafeln für Kellerautomaten entwickelt hat, oder die Programmierung durch Entscheidungstabellen auf der Grundlage der Booleschen Algebra.

Die Synthese von Sprachfixierung und Zerteilerkonstruktion im Compilerbau bewirkt, daß nun keine Mißverständnisse über den Umfang einer Programmiersprache mehr möglich sind. Die Frage der Zugehörigkeit eines Programmes zur Wortmenge aller gültigen Programme einer Sprache wird durch ein kombinatorisches System, eine formale Grammatik zu dieser Sprache entschieden. Der daraus erzeugte Zerteiler entscheidet nicht anders. Man kann sagen, er ist diese Grammatik. Seine Arbeitsweise, rekursiver Ab- und Aufstieg, ist klar und durchschaubar, aber nicht trivial. Rekursives Programmieren ist eines der Kennzeichen nichtprozeduraler Programmierung, mit der sich Abschnitt 3.1 noch befaßt.

Ein einfaches Beispiel für die Identität von Definition und Analyse ist folgender Text:

IDENTIFIER :

LETTER, REST OF IDENTIFIER.

REST OF IDENTIFIER :

LETTER, REST OF IDENTIFIER;

DIGIT, REST OF IDENTIFIER;

ϵ .

(ϵ = leere Zeichenkette). Es handelt sich um zwei Regeln aus dem Produktionensystem einer Grammatik, welche die bekannte Bildung von Bezeichnern festlegen. Sie sind in der van Wijngaarden-Notation dargestellt. Derselbe Text ist aber auch Teil einer Zerteiler-Beschreibung und sagt aus; Es gibt Prozeduren, die in der eingegebenen Zeichenkette einen Buchstaben erkennen und verarbeiten, das Postfix eines Bezeichners oder den ganzen Bezeichner. Das Kombinieren der Prozeduraufrufe geht aus den rechten Seiten der Regeln hervor; sie bilden Prozedurkörper. Die Prozedurbezeichnungen sind durch die linken Regelseiten gegeben. Terminale Symbole der Grammatik, etwa LETTER oder LETTER A SYMBOL, besitzen einen Prozedurrumpf, der einen Symbolvergleich enthält und das gelesene Zeichen eventuell wegstreicht. Der Zerteiler ist ein genaues Abbild der *parsing machine* von Knuth.

Die Leistung des CDL-Compiler-Compilers ([Ko 71b]) ist es nun, daß er eine Zerteiler-Beschreibung in der Art des obigen Beispiels mechanisch überträgt in eine Folge von Prozeduren, die er in einer Wirtssprache formuliert. Je nach Compiler-Version handelt es sich dabei um eine problemorientierte Programmiersprache (Algol 60, Algol W, PL/1) oder um eine Assemblersprache (IBM 370, Siemens 4004, CDC 6500, CDC 3300, TR 440, PDP 10, PDP 15). Der Compiler-Compiler erstellt aus einer einstufigen van Wijngaarden-Grammatik den Zerteiler, generiert in der gewählten Wirtssprache. War die Grammatik vom Typ LL(1), so arbeitet nun der Zerteiler nach diesem Verfahren. Aber er leistet eben nur Syntaxanalyse und löst nur das Wortproblem. Die weiter-

ren Compilerarbeiten vom Listenführen über semantische Untersuchungen bis hin zur Codeerzeugung sind dadurch noch nicht beschrieben. Darum wurde die Compiler-Beschreibungssprache erweitert. Die Grundlage von CDL sind die van Wijngaarden-Schreibweise kontextfreier Grammatiken und die Affix-Grammatiken von Koster ([Ko 71a]). Hinzu kommen die Möglichkeiten der Definition von Makrobefehlen, von semantischen Aktionen und der Deklaration von Daten. Abgerundet wird diese Sprache durch einigen "syntaktischen Zucker" wie lokale Sprünge, Gruppierung von Alternativen und den Anschluß von externen Daten oder Funktionen (s. [Ko 71b]). Ist eine Syntaxdefinition als van Wijngaarden-Grammatik gegeben, so wird dieser Text zu einer vollständigen Compilerbeschreibung erweitert durch "Hineinfüllen" (Implementieren) aller zur Syntaxanalyse hinzukommenden Compilerleistungen. Ein Beispiel für eine Compilerbeschreibung in CDL ist der Compiler-Compiler selbst (z.B. in [Br 74]). Eine CDL-Beschreibung ist noch verständlicher und besser lesbar als etwa ein Algol- oder Pascal-Programm.

Die Identität von Syntaxdefinition und -analyse bringt eine große Sicherheit in die Arbeit eines Compilers. Der Nachweis einer korrekten Syntaxanalyse entfällt. Die Verwendung von CDL sichert darüber hinaus eine hervorragende Strukturierung und einfache Wartung des Compilers, ebenso hohe Portabilität. Im folgenden Kapitel 3 wird untersucht, ob und wie weit sich diese Merkmale auf große Programmsysteme auch außerhalb des Bereiches Compilerbau und Compilererzeugung übertragen lassen.

3 CDL, EIN BEITRAG ZUR PROGRAMMIERMETHODIK

Will man die Errungenschaften der grammatikgesteuerten Syntaxanalyse auf die allgemeine Programmierung im technisch-wissenschaftlichen Bereich übertragen, so muß man für die zentralen Betriffe wie *Grammatik*, *Zerteiler* und *Produktion* eine Entsprechung etablieren. Für *Grammatik* wird der Begriff *Problembeschreibung* gesetzt, für *Zerteiler* der Begriff *Problemlösung* und für *Produktion* schließlich *Teil der Problembeschreibung*. Gelten diese Gleichsetzungen, so ist die Identität von Problembeschreibung und Problemlösung erreicht, wie in Abschnitt 2.3 für die Syntaxanalyse beschrieben. Insbesondere wirft der Begriff "Teil der Problemlösung" ein Licht auf den Entstehungsprozeß und die Vorgehensweise bei der Lösungsfindung.

3.1 Problembeschreibung

Die Entwicklung der Programmiersprachen führt weg von der Anweisungsebene und hin zur Problemebene. Diese etwas vage Aussage wird von den Zielen, die im Zusammenhang mit "sehr hohen" Programmiersprachen (very high level, VHL-Sprachen) genannt werden, nicht viel klarer ausgedrückt. Ein VHL-Programm soll darstellen, was für ein Problem gelöst wird, und weniger, wie dieses Problem gelöst wird ([Le 74a]). Digitale Rechenanlagen arbeiten überwiegend noch sequentiell, und diese Vorstellung ist bei allen höheren Programmiersprachen sichtbar. Die Abstraktion von der Ebene einer Maschinenarchitektur mit Registern und einem Befehlsvorrat zur Ebene der Programmarchitektur mit Blöcken, Programmeinheiten und Sprachkonstrukten war ein großer Schritt, der Übergang von höheren Programmiersprachen zum nichtprozeduralen Programmieren muß ein nächster Schritt innerhalb der genannten Entwicklung sein.

Auch für den Begriff *nichtprozedurale Programmierung* gibt es keine allgemeingültige Definition. Grundsätzlich handelt es sich um die Unterdrückung unnötiger Einzelheiten bei der Formulierung von Algorithmen. Leavenworth schreibt in

[Le 74b]: "Nichtprozedurales Programmieren weist viele Ziele der strukturierten Programmierung auf; Es geht um die Konstruktion von Programmen, die leichter zu verstehen sind, zu modifizieren und zu testen." In dieser Arbeit werden drei Kennzeichen nichtprozeduralen Programmierens genannt:

- rekursives Programmieren
- kombinatorisches Programmieren und
- nichtdeterministisches Programmieren,

Rekursives Programmieren wird bereits von den höheren Programmiersprachen unterstützt. Es erleichtert sehr das Handtieren mit baumartig strukturierten Daten und allgemein mit rekursiven Datenstrukturen (s. auch [DDH 72], Seite 142). Zum kombinatorischen Programmieren wird in [Le 74b] ausgeführt: "Die Idee beim kombinatorischen Programmieren ist es, Funktionen zu manipulieren und zu kombinieren, um dadurch Schleifen, bedingte Anweisung und rekursive Aufrufe größtenteils zu eliminieren." Durch Hintereinanderschalten werden Funktionen gebildet, welche einfache Funktionen auf Listen und Mengen abbilden und dabei auch gewisse Mengenoperationen leisten. Beim nichtdeterministischen Programmieren hat der Benutzer die Möglichkeit, Algorithmenteile mit dem Aufruf einer Wahlfunktion CHOICE (N) einzuleiten. Sie bestimmt zur Laufzeit einen Wert aus dem angegebenen Bereich und damit konkrete Daten aus einer Menge von Alternativen, für deren Gesamtheit die folgenden Algorithmenteile einmal formuliert sind. Hinzu kommen zwei Funktionen für "Erfolg" oder "Fehler". Die eine markiert das normale Ende des Algorithmus', die andere veranlaßt zur Laufzeit das Zurückgehen im Algorithmus zur jeweils letzten Wahlmöglichkeit und die wiederholte Ausführung mit anderen Daten. Der interne Verwaltungsaufwand, den nichtdeterministisches Programmieren zur Laufzeit verursacht, ist mit dem eines Suchalgorithmus' in Labyrinthen vergleichbar. Das Beispiel, an dem in [Le 74b] nichtdeterministisches Programmieren demonstriert wird, ist ein nach dem LL(1)-Verfahren arbeitender Zerteiler.

Rekursiv und nichtdeterministisch ist die Definition einer formalen Sprache durch eine formale Grammatik. Diese bei [Le 74b] genannten Kennzeichen nichtprozeduralen Programmierens sind einer formalen Grammatik immanent. Das "Problem", das sie beschreibt, ist eine i. allg. unendliche Wortmenge. Ein Anwenderproblem kann grundsätzlich ebenso beschrieben werden. Der unendlichen, durch ein endliches System erzeugten Wortmenge entspricht dann die potenziell unendliche Menge von Problemstellungen, die sich durch konkrete Daten individuell darstellen, durch einzelne Programmläufe jeweils gelöst werden, die aber alle durch einen einzigen Algorithmus behandelt werden.

3.2 Problemlösung

Die Programmiersprache CDL ist nicht nur eine Compiler-Beschreibungssprache. Zwar wurde sie bisher ausschließlich im Compilerbau eingesetzt und bei der mehr oder weniger automatischen Erzeugung von Compilern, ihre Vorzüge müssen aber nicht auf den Einsatz in diesen Gebieten beschränkt bleiben. Liegt die Problembeschreibung in CDL vor, so ist sie vermöge der Übertragung durch den CDL-Übersetzer in eine Wirtssprache bereits identisch mit der Problemlösung. CDL-Programme unterscheiden sich von kontextfreien Grammatiken in van Wijngaarden-Notation nur durch Affixe und die in Abschnitt 2.3 erwähnten Möglichkeiten der Sprache. So ist die Verwendung direkt oder indirekt rekursiver Funktionen immer möglich. Da die Übergabe von Funktionen als Parameter beim Aufruf von Funktionen aber nicht erlaubt ist, handelt es sich immer um statische Rekursivität wie in Grammatiken. Deswegen ist auch kombinatorisches Programmieren im Sinne von [Le 74b] nicht möglich, sondern nur Kombinatorik im Sinne des Hintereinanderausführens von Ableitungsschritten, welcher das Schachteln von Funktionsaufrufen entspricht. Die Technik des *backtracking* als notwendige Möglichkeit beim nichtdeterministischen Programmieren wird in CDL unterstützt. Allerdings ist eine nicht-deterministische Syntaxanalyse in Compilern nicht praktikabel.

Darum wird auch in CDL keine Wahlfunktion bereitgestellt. Alternative Verzweigungen der Algorithmen werden in der angegebenen Reihenfolge durchlaufen. Zur Verdeutlichung zeigt Fig. 2 die Auflösung einer CDL-Programmeinheit, genannt "Regel", in Alternativen und Einzelschritte ([Ko 71b]).

ALGORITHMUS XYZ :

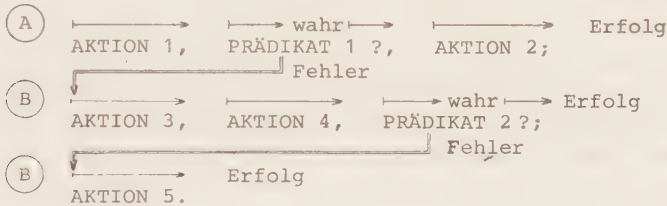


Fig. 2: Beispiel für *backtracking* in CDL

Die Funktionen für "Erfolg" und "Fehler" sind implizit vorhanden. Das Zurückgehen innerhalb der Alternativen im Fehlerfall ist dabei nicht problematisch. Schwierigkeiten bereiten aber immer Seiteneffekte, die im Beispiel etwa bei der Ausführung der Aktionen 1, 3 oder 4 aufgetreten sein könnten. Da die ganze über Syntaxanalyse hinausgehende Verarbeitung eines Programmes als "Seiteneffekt" betrachtet werden kann, wird auch deutlich, warum Compiler deterministische Analyseverfahren benutzen. Ein Seiteneffekt wird in CDL-Programmen jedoch kompensiert, nämlich das Weiterlesen in der Eingabezeichenkette. Wenn der Benutzer es wünscht und seine Eingabedaten mit direktem Zugriff bereitstellt, dann werden vom CDL-Übersetzer geeignete Anweisungen eingeschoben, die einen Lesezeiger manipulieren. Er wird zuerst in einer lokalen Größe abgelegt (Fig. 2, Stelle (A)) und bei Bedarf immer wieder auf die alte Stelle in der Zeichenkette zurückgesetzt (Stellen (B)). Auf diese Weise ist mit CDL-Programmen sogar Syntaxanalyse mit Rücksetzen (engl. back-up) möglich ([Ko 71b], [Sn 75a]).

Bei der Syntaxanalyse schlugen sich die Symbole der zugrundeliegenden Grammatik im Zerteiler nieder in Form von Booleschen Funktionen, genannt "Prädikate" oder "Teste". Der Zerteiler war das Fachwerk für den ganzen Compiler. Bei allgemeinen Problemlösungen ermöglicht die Verwendung von Prädikaten und Testen auch eine problemidentische Darstellung der logischen Struktur. Es ist verblüffend, wie gut das Verhalten arbeitender CDL-Programme der ursprünglichen, informalen Problemdarstellung entspricht. Das liegt an dem Übergang von der informalen zur formalen Problembeschreibung. Bei Verwendung von CDL als Implementierungssprache ist dieser Schritt klein, einfach und rasch zu vollziehen.

3.3 Der Prozeß der Lösungsfindung

Von den Verfahren der Syntaxanalyse haben sich zwei Begriffe auf den Bereich der Programmiermethodik übertragen, der "top-down-Entwurf" und die "bottom-up-Konstruktion" großer Programmsysteme. Es ist ungewiß, welche der beiden Vorgehensweisen der Kreativität eines Programmierers mehr entgegenkommt. Sicher ist es nicht nur eine Frage der Gewohnheit. Beim top-down-Entwurf ist die Programmumgebung an den Aufrufstellen einer Prozedur schon bekannt, ehe die Prozedur selbst definiert wird, bei der bottom-up-Konstruktion aber nicht. Sicher ist aber auch, daß man beide Entwurfsarten nicht für sich alleine betrachten darf, weil man sonst zu folgenden kuriosen Situationen gelangt. Dies gilt sowohl für den Entwurf von Algorithmen wie auch für die Beschreibung von Bildern im Dialog (Kapitel 4).

3.3.1 Der top-down-Entwurf

Bei dieser Programmiermethode gelangt man zu einem Ergebnis, das man als Programmbaum darstellen kann (s. u. Fig. 3). Der Entwurfsprozeß beginnt bei dem höchsten Knoten, einer Bezeichnung des ganzen Problems. Die Aufteilung eines Knotens

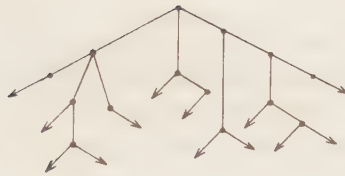


Fig. 3: Programmbaum

in Äste entspricht der Zerlegung in Teilprobleme, welche den Knoten ersetzt. Die Blätter des Baumes sind einfache Anweisungen in einer Programmiersprache. Das Ergebnis liegt als Programm vor, wie es die Folge der Blätter liefert, in dem die Bezeichnungen der Knoten vielleicht noch als Kommentare enthalten sind. In diesem Programm sind alle Teilprobleme unabhängig voneinander gelöst. Sie sind optimal gelöst, aber möglicherweise auch mehrfach, denn das Bilden von Unterprogrammen wird bei diesem Vorgehen nicht erzwungen. In konsequenter Fortführung dieser Methode gäbe es nicht einmal die Verwendung von Benutzerbibliotheken, von Standardfunktionen oder Betriebssystemfunktionen, denn diese Hilfen stellen eine Unterprogrammtechnik in tieferen Ebenen dar. Stattdessen wären alle Leistungen im Programm selbst enthalten. Querverbindungen gäbe es nur durch gemeinsame, globale Daten. Nicht einmal der Zugriff auf Daten, ihre Verwaltung und ihr Schutz wären zentral geregelt.

3.3.2 Die bottom-up-Konstruktion

Programmbausteine oder programmierte Verfahren seien schon vorhanden. Dabei kann es sich um einzelne Befehle, um Makrobefehle, um ganze Unterprogramme in irgendeiner Programmiersprache oder um Datenmoduln mit Daten und zugreifenden Algorithmen handeln. Fertige Teillösungen für häufig auftretende Probleme können in Bibliotheken, Programmpaketen oder Methodenbanken hinterlegt sein. Nun werden solche Bausteine,

sofern vereinbar, frei kombiniert, man erhält komplexere Bausteine und sucht nach Problemen und Anwendungen, bei denen die verfügbaren Algorithmen nützlich sein könnten. Auch diese Vorgehensweise ist für sich allein fragwürdig. Praktisch einsetzbar sind nur die Mischformen beider Entwurfsmethoden.

3.3.3 Synthese beider Methoden

Theoretisch ist es möglich, ein Programm ausschließlich im top-down-Entwurf zu erstellen. Die Programmstruktur wäre dann sogar extrem problemnah. Obwohl kein Mensch so vorgehen würde, wollen wir doch ein streng nach dieser Art entstandenes Programm betrachten. In einem zweiten Arbeitsgang könnte man nun funktionale Ähnlichkeiten verschiedener Programmteile suchen und dafür Unterprogramme einführen. Dieser Prozeß sollte an der Basis der Programmpyramide beginnen und sie jeweils auf ihrer ganzen Breite durchsuchen. Völlig identische Programmteile wird man dabei nur selten finden, meistens müssen sie aneinander angepaßt werden. Diese Anpassung geschieht teilweise im Prozedurrumpf, teilweise in der Umgebung der Aufrufe. Hier wird deutlich, warum dieser Vorgang der Verschmelzung von Programmteilen nicht zu früh erfolgen darf und eigentlich erst nach Fertigstellung des gesamten top-down-Entwurfes beginnen sollte. Sonst werden nämlich nicht alle Kandidaten zu einer möglichen Verschmelzung erfaßt, und die Verschmelzung erfolgt einseitig. Später hinzukommende Programmteile müssen dann zu sehr "verbogen" werden oder bleiben eben unberücksichtigt. Der Vorgang einer Verschmelzung geeigneter Programmteile ist ein heuristischer Prozeß. Die endgültige Struktur und Qualität eines Programmes hängt sehr stark davon ab. Falsch wäre es hierbei, nur Zeichenketten zu parametrisieren und zu verschmelzen, weil die Struktur und Verständlichkeit eines Programmes sehr darunter leiden, wenn nicht semantisch abgeschlossene Programmteile herausgegriffen werden. Das

Risiko bei aller Verschmelzung liegt darin, daß dieser Vorgang sehr bald einsetzt und eine vollständige Verfeinerung des ganzen Programmes meistens fehlt.

Auch in den Mischformen beider Entwurfsmethoden ist immer eine Entwurfsrichtung festzustellen. Die eine Methode überwiegt, sie ist von der anderen nur beeinflusst. Beim top-down-Entwurf prüft man ständig die Möglichkeiten für den Einsatz von Unterprogrammen. Auch kennt man die Hilfen von Programmbibliotheken und des Betriebssystems, die man einsetzen will, man steuert beim Entwurf darauf zu und versucht eine "glatte Landung", den zwanglosen und wirkungsvollen Einsatz der gebotenen Unterstützung.

Im umgekehrten Fall geht man von vorhandenen Bausteinen aus. Der Konstruktionsprozeß ist beeinflusst von einer Zielvorgabe. Bei konkreten Projekten, die als Einzellösung geplant werden, ist so eine Zielvorgabe umfangreich und detailliert, wie etwa bei dem Laufzeitsystem zu einer Programmiersprache. Wendet man sich dagegen mit einem Dienstleistungs-Programmpaket an eine Anwendergruppe, so ist man statt dessen mit einem breiten Aufgabenspektrum aus dieser Gruppe konfrontiert. Die Umkehrung des obengenannten Verschmelzens ist nun die Parametrisierung von Prozeduren und Moduln und die Anpassung von Lösungsmethoden an konkrete Aufgaben.

Die Synthese beider Programmiermethoden liegt nun im abwechselnden Einsatz der beiden beeinflussten Verfahren. Das zeitlich versetzte top-down-Entwerfen und bottom-up-Korrigieren einer Programmstruktur wird darum auch als *yo-yo-programming* bezeichnet. Es hat den Anschein, als ließen sich projektspezifische Eigenschaften und der globale Programmentwurf besser in der top-down-Richtung verwirklichen, während Lösungsverfahren für Teilprobleme, die in ähnlicher Form häufig in Erscheinung treten, besser als bottom-up-

gewachsene Programmteile in die Lösung integriert werden.
In praktischen Ergebnissen kommt Holager in der Arbeit [Ho 75].

3.3.4 Lösungsprozess bei Verwendung von CDL

Das Produktionensystem einer kontextfreien Grammatik enthält vielfach Ersetzungsvorschriften mit gleicher linker Seite, die dann in einer sogenannten Produktions-Regel zusammengefaßt werden. Sie definiert ein Zwischensymbol der Grammatik durch eine Reihe von alternativen Ersetzungen, deren jede in eine Sequenz einzelner Zeichen zerfällt. Völlig analog wird in einer CDL-Beschreibung durch jede Regel die Zerlegung eines Programmknocens definiert als alternative Folgen von Einzelschritten, deren Abarbeitungsreihenfolge durch die impliziten Funktionen für "Erfolg" oder "Fehler" innerhalb jeder Regel gesteuert wird. Am Beispiel wurde das in Abschnitt 3.2 gezeigt, ausführliche Darstellungen stehen in [Ko 71b] und [De 75]. Einfache Beispiele für CDL-Regeln sind

CDL BESCHREIBUNG :

```
REGEL 2, CDL BESCHREIBUNG;  
ENDE DER BESCHREIBUNG ?;  
ÜBERLIES BIS PUNKT, CDL BESCHREIBUNG.
```

und

SLR TABLE CONSTRUCTOR :

```
START SYSTEM  
READ & IF GRAMMAR, PRINT NONTERMINALS, PRINT TERMINALS,  
TEST FOR EMPTY NONTERMINALS, PRINT EMPTY NONS,  
SET FOLLOWER SETS, PRINT RECURSIVE NONS,  
CONSTRUCT THE ACTION TABLE, PRINT ACTION TABLE,  
LIST WORDS.
```

Das erste Beispiel beschreibt die Übersetzung einer Folge von Regeln, die explizit abgeschlossen wird. Das zweite Beispiel enthält nur eine einzige Alternative mit vielen Einzelaktionen.

Die Entstehungsreihenfolge der CDL-Regeln als Einzelteil der Problembeschreibung spiegelt zugleich die Struktur wieder, in welcher der Strukturbaum der Lösung entsteht. Beide Methoden der Entwurf von oben nach unten und *ungerichtet*, sind bei Verwendung von CDL anwendbar, aber der top-down-Entwurf wird schon allein durch die äußere Form der CDL-Regeln suggeriert.

Den Programmknotten entsprechen bei CDL die Regelbezeichnungen. Die in den Alternativen enthaltenen Bezeichnungen stehen für die Zweige im Programmbaum. Keine Blätter sind Bezeichnungen, die durch Anweisungen in der Wirtssprache oder außerhalb des Programmes definiert sind. Durch das hierarchische Zerlegen von Algorithmen leisten CDL weitgehend das, was von sehr hohen Programmiersprachen allgemein gefordert wird. Bei der Dekomposition eines Algorithmus wird für seine Teile zunächst nur festgelegt, was für Teilprobleme sie lösen, nicht wie das in einzelnen geteilt. Die Algorithmenteile erhalten ausführliche Bezeichnungen, deren Lesbarkeit und Verständlichkeit sich auf die ganze Problembeschreibung übertragen. Bei späteren Verfeinerungsschritten werden nun für die Teilalgorithmen prozedurale Zerlegungen gebildet. Dadurch können für jedes Teilproblem die Fragen nach dem "Was?" und "Wie?" zeitlich getrennt beantwortet werden, und das ist eine große Hilfe beim Entwickeln einer Problemlösung.

3.4 Eigenschaften von CDL-Programmen

Eigenschaften von Programmen resultieren aus den Merkmalen der verwendeten Sprache und ihrer Realisierung auf einem Rechner. CDL ist eine Problembeschreibungssprache mit einigen Merkmalen von sehr hohen Programmiersprachen. Der Schritt von einer informellen Problembeschreibung zur Formalisierung ist klein. Da die Formvorschriften für CDL-Beschreibungen sehr einfach sind, ist dieser Schritt auch schnell und einfach zu vollziehen. Die völlige Entsprechung des qualitativen Laufverhaltens mit der ursprünglichen Problemstellung ist bei allen CDL-Programmen

feststellbar. Eine CDL-Beschreibung enthält voneinander abgegrenzte, klar überschaubare funktionale Einheiten. Dadurch ergeben sich auch in der ganzen Problemhierarchie klare Schnittstellen. Die lateinische Faustregel *divide et impera* (teile und verführe) ist in CDL konsequent verwirklicht. Folge davon ist die Möglichkeit einfacher Arbeitsteilung, sowohl innerhalb eines Teams als auch für den einzelnen Mitarbeiter, was beides eine rasche Implementierung begünstigt. Die Zerlegung von Programmen in alternative und sequentielle Schritte ist einerseits so einfach, daß die logische Struktur der fertigen Fortsetzung unmittelbar erkannt wird, andererseits aber so mächtig, daß alle Programmstrukturen, die von der strukturierten Programmierung her bekannt sind, sich unmittelbar darstellen lassen. Würden CDL-Regeln in eine Textzeile passen, so wären Zeilenkennungen für die Ablage überflüssig. Ein System zur CDL-Programmierung im Dialog könnte stattdessen die Regelbezeichnung verwenden. Bei EBIOS werden Bildbeschreibungen in Zeilen unterteilt (vgl. Abschnitt 4.3.1.3). Hier sind "nichtterminale" Bildnamen zur Kennzeichnung von Zeilen denkbar, aber nicht ausreichend (Abschnitt 4.3.2.3). So wurde an den bei Dialogsprache bekannten numerischen Bezeichnungen für ausführbare Einheiten festgehalten.

Die hierarchische Struktur der ganzen CDL-Beschreibung mit allen Schnittstellen ermöglicht auch ein abgestuftes, hierarchisches Verständnis beim späteren Durchlesen. Für systematische und teilweise Programmteste ist eine CDL-Beschreibung sehr gut geeignet. Die Übertragbarkeit von CDL-Programmen ist bekannt: gegeben wie die von Programmen in beliebigen höheren Programmiersprachen, wenn auf beiden beteiligten Rechenanlagen der gleiche Übersetzer verfügbar ist. Schwierigkeiten ergeben sich dabei häufig durch unterschiedliche Wortlängen, verschiedene Textzeilenanordnungen und -verarbeitung bei den beteiligten Rechnern oder ähnliche Gegebenheiten. Und diese Schwierigkeiten treten dann meistens bei einfachen, elementaren Programmmanövern auf. Die relativ maschinennahen Tätigkeiten und Befehle werden aber bei CDL-Beschreibungen immer als

Makro-Vereinbarungen definiert und durch Elemente der Wirtssprache ausgedrückt. Durch Ändern allein dieser Makro-Vereinbarungen kann also ein CDL-Programm den veränderten Gegebenheiten beim Zielrechner angepaßt werden. Notwendig ist das Austauschen der Makro-Vereinbarungen natürlich, wenn der CDL-Übersetzer der Zielanlage eine andere Wirtssprache verwendet. Diese Ersetzung ist ein einfacher, mechanischer Arbeitsgang, der für jede Vereinbarung unabhängig geschieht und keinerlei globale Programmkenntnis erfordert. Die Möglichkeiten für das Übertragen von CDL-Programmen erhöhen sich gegenüber denen für sonstige problemorientierte Sprachen, weil die Übertragung nicht nur auf der CDL-Ebene geschehen kann, sondern auch auf der Ebene der Wirtssprache.

Eine der Vorarbeiten für die Implementierung der Dialogsprache EBIDIS auf dem Graphikrechner des Institutes war das Bereitstellen von CDL als Implementierungssprache. Eine Übersetzer-version mit Algol 60 als Wirtssprache stand zwar schon zur Verfügung, aber die von diesem Übersetzer erzeugten Algol-Objekte hatten hohe Laufzeiten. Darum entstand eine andere Version des CDL-Übersetzers, welche die Assemblersprache des Graphikrechner, einer Anlage PDP-15, als Wirtssprache benützt. Dieser Übersetzer für CDL 15 wurde auf dem Großrechner der Informatik in CDL geschrieben, durch Selbstübersetzung auf dem Großrechner transportfähig und durch Selbstübersetzung auf dem Graphik-Rechner (*bootstrapping*) einsatzfähig gemacht. Dieser Übersetzertransport demonstrierte das einfache und zuverlässige Vorgehen beim Übertragen von CDL-Programmen. Im einzelnen wurden dazu die in Figur 4 dargestellten Schritte unternommen.

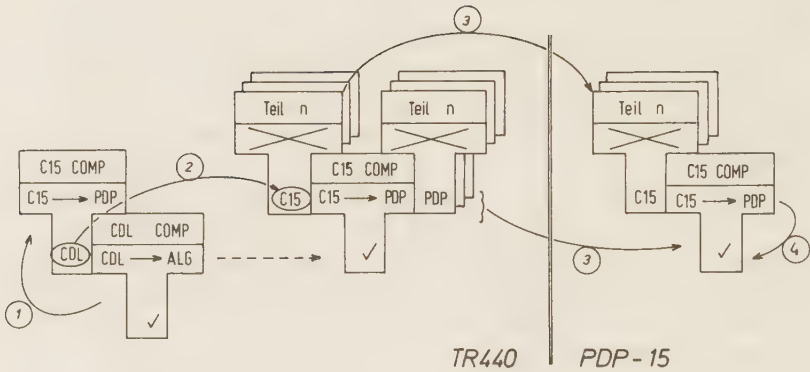


Fig. 4: Übertragung eines Übersetzers

- Schritt ①: Die CDL-Beschreibung des CDL-15-Übersetzers (C15 COMP) entsteht unter Verwendung der CDL-Beschreibung des CDL-Übersetzers (CDL COMP).
- Schritt ②: Der CDL-15-Übersetzer wird für die Zielanlage segmentiert durch Aufteilen der CDL-Beschreibung und Kopieren einiger Deklarationen. Gleichzeitig ersetzen Makrodefinitionen in PDP-Assembler die bisherigen in Algol 60.
- Schritt ③: Die CDL-15-Quellen der Übersetzersegmente und die durch Selbstübersetzung erhaltenen Assembler-Quellen werden zur Zielmaschine übertragen.
- Schritt ④: Nach dem Assemblieren und Binden der Übersetzersegmente auf dem Graphikrechner arbeitet der CDL-15-Übersetzer genauso wie zuvor auf dem Großrechner. Für die weiteren Selbstübersetzungen (*bootstrapping*) wird nur noch der Zielrechner benötigt.

Die durch Makrodefinitionen gegebene Flexibilität von CDL-Programmen ist natürlich nicht nur beim Übertragen zwischen verschiedenen Rechenanlagen von Vorteil. Änderungen bei ein und derselben Anlage, etwa durch Ersetzen oder Hinzunahme peripherer Geräte, werden durch entsprechende Modifizierung der Makrokörper noch leichter berücksichtigt. Durch Verwendung einer Assemblersprache als Wirtssprache ist der Anschluß von Codeprozeduren sehr einfach. Weiterhin ist mit den Assembler-Makrodefinitionen in einer CDL-Beschreibung immer eine flexible Schnittstelle zum Betriebssystem der Anlage gegeben, wie das bei Implementierungssprachen auch nötig ist.

Bisher wurden nur Vorzüge bei der Verwendung von CDL genannt. Der Preis für die Problemnähe von CDL-Beschreibungen ist ein erhöhter Bedarf an Hauptspeicher und Laufzeit gegenüber handgeschriebenen Programmen in der Wirtssprache. Das ist eine grundsätzliche Erfahrung beim Einsatz von höheren Programmiersprachen. Diesem Problem kann aber hier wie dort durch optimierende Compiler begegnet werden. Die Verwendung von Algol 60 als Wirtssprache bedingt große Laufzeiten, weil dann nichtspezifizierte Prozedurparameter so aufwendig behandelt werden. Diese Schwierigkeit hat man bei der Verwendung einer Assemblersprache nicht. Trotz ständiger Kontrollen bei der Übergabe von Prozedurparametern u. ä. rechnen CDL-15-Programme so schnell, daß CDL 15 für die Bearbeitung von Echtzeitaufgaben geeignet ist. Auch das ist eine Eigenschaft, die von Sprachen zur Systemimplementierung gefordert wird, insbesondere bei der Implementierung von Dialogsystemen.

4 BILDPROGRAMMIERUNG IM DIALOG

4.1 Dialogischer Einsatz von Rechenanlagen

Der Begriff *Dialog* ist zum eindrucksvollen Modewort geworden. Rechensysteme, die den Benutzer die Mühsal des Informationsaustausches etwas vergessen lassen dadurch, daß sie ihren Ein- und Ausgabeverkehr angenehm und hilfreich gestalten und dem Benutzer die Ahnung eines Gespräches mit einem intelligenten Partner vermitteln, solche Rechensysteme werden beizeiten mit dem Attribut "Dialog-" versehen. Das gilt für alle Anwendungen, seien es Übersetzer für höhere Programmiersprachen, seien es Kommandosprachen in Betriebssystemen, Informationssysteme oder sonstige Datenverarbeitung.

Dabei setzt doch ein Dialog das Vorhandensein wenigstens zweier gleichberechtigter Partner voraus, die "durch das Gespräch" (*διὰ λόγον*) sich gegenseitig bereichern und vorwärtsbringen. Bei zwei menschlichen Partnern hat jeder sein Hintergrundwissen und gute Ideen, jeder hat seine eigene Sicht und Darstellung der Verhältnisse. Eine Rechenanlage als Dialogpartner wird nun zunächst mehr die Funktion der Wissensspeicherung und -darstellung übernehmen, der menschliche Partner wird Zusammenschau und Integration von Wissen und seine Kreativität in den Dialog einbringen. Damit ist die Symmetrie des Dialoges wegen der Verschiedenartigkeit beider Partner zunächst gestört. Trotzdem müssen beide Partner den Dialog fördern und vorantreiben. Dazu gehört wenigstens, daß ein Dialogsystem den menschlichen Partner nicht behindert.

Ein kurzer Seitenblick auf die Rolle des menschlichen Benutzers bei stapelorientierten Arbeiten an Rechenanlagen läßt die Mühen einer Pionierzeit erkennen. Der Anwender leistet zunächst einen riesigen Beitrag zur Verständigung: Er lernt Programmiersprachen und Kommandosprachen. Dann wird ihm auch die ganze Denkarbeit aufgebürdet: Er muß Problemlösungen und Algorithmen entwickeln und abbilden auf die Ebenen der erlernten

Sprachen. Dann wird er bestraft durch abgebrochene Aufträge und obskure Fehlermeldungen. Und erst, wenn er seine Hoffnung auf baldige Resultate ersetzt durch Ausdauer und Starrsinn, wird ihm ein Fortschritt gelingen. Gute Ideen entstehen nicht, wenn die Arbeit in vielen mühsamen Einzelheiten erstickt. Die Denkpausen, welche beim Stapelbetrieb von Auftragsabgabe bis zur Einsicht in etwaige Ergebnisse vergehen, sind nicht so schlimm. Daß solche Denkpausen bei Arbeiten im Dialog fast wegfallen, ist nicht unbedingt ein Gewinn. Denn eine ständige Eingabeanforderung und der Zwang zum Denken und Handeln, auch der hohe Geräuschpegel in manchen Räumen mit Rechenanlagen können einen Benutzer wohl belasten. Aber die Last der Verständigung bei der Formulierung von Aufträgen an einen Rechner und bei der Fehleranalyse, diese Last sollte nicht der Benutzer tragen, sondern das Rechensystem. Das weist auch auf die Bedeutung einer Dialogsprache hin, derer sich beide Partner bedienen. Sie ist Vehikel und auch Antrieb in einem Dialog, was in Abschn. 4.3.2 noch erörtert werden wird.

Die Verschiedenartigkeit beider Partner in einem Mensch-Maschine-Dialog ist kein bedauerliches Übel, das hingenommen werden muß. Sie ist zum Teil überbrückbar. Arbeiten auf dem Gebiet der künstlichen Intelligenz und des beschränkten Einsatzes natürlicher Sprache lassen das hoffen. Die Formulierung eines Problems in natürlicher Sprache und das selbständige Suchen und Erarbeiten einer Lösung durch das System sind das Fernziel beim Einsatz intelligenter Rechensysteme. Die Eingabe und Auswertung von Sätzen in natürlicher Sprache wird bereits praktiziert, z.B. als Abfragesprache bei Datenbanken ([Kog 76], [KräLo 74], [GuLö 77]), nichtprozedurales Problemlösen setzt eine gewisse Abkehr von prozeduralen Programmiersprachen voraus. Das Arbeiten im Dialog mit seiner freien Fortsetzbarkeit ist dazu grundsätzlich geeignet. Es gestattet z.B. unmittelbare Entscheidungen des Benutzers und die Auswahl von Algorithmen etwa für die Aufbereitung eines Anwendungsproblems, für die Berechnung einer Lösung und für die Darstellung der Lösung im Problembereich. Weitere nichtalgorithmische, also heuristische Tätigkeiten des

Benutzers sind das erstmalige Erstellen eines Algorithmus' oder das Anfertigen einer formalen Problembeschreibung durch nicht-prozedurale Formulierungen. (Textaufgaben der Schulzeit waren informale, nichtprozedurale Problemstellungen). Diese Tätigkeiten werden in [KuWi 75] etwas ausführlicher beschrieben.

Auch die folgenden Beispiele für die Rechnerunterstützung im nichtalgorithmischen Bereich durch Dialogsysteme findet man bei [KuWi 75]. Die Dialogsprache NAPSS (Numerical Analysis Problem Solving System) z.B. gibt dem Benutzer folgende Möglichkeiten:

- a) Er schreibt eine Prozedur zur Errechnung der Lösung.
- b) Er definiert das Problem und spezifiziert, welches der dem System bekannten Lösungsverfahren zur Lösung herangezogen werden soll.
- c) Er definiert das Problem und überläßt die Auswahl eines der verfügbaren Lösungsverfahren dem System.

Ein Gleichungssystem ist eine nichtprozedurale Beschreibung eines Anwendungsproblems. Die Eingabe

GL.1: $2 \sin(2X) = Y * \exp(X)$

GL.2: $2 \cos(2Y) = X * \exp(-Y)$

z.B. definiert in NAPSS ein Gleichungssystem mit Namen "GL". Die SOLVE-Anweisung von NAPSS hat die allgemeine Form

SOLVE <Gleichungen>, FOR <Variable> [, <Zusätze>]

Beispiele für SOLVE-Anweisungen sind

SOLVE GLSYS, FOR X, Y, USING NEWTON METHOD

SOLVE A * X = LAMBDA * X, FOR LAMBDA, X

Die Zusätze betreffen Angaben über Wertzuweisungen, Anzahl von Lösungen, Methodenauswahl, Rechengenauigkeit u.a. ([KuWi 75] S.106).

Bei einer anderen Dialogsprache, VENUS, werden die Äußerungen des Benutzers nicht als Befehle für das System aufgefaßt, sondern als Mitteilungen oder Fragen. Das System stellt seinerseits Fragen oder es gibt Antworten. Auf Fragen des Systems braucht nicht sofort geantwortet zu werden. Es versteht dann auch sonstige Eingaben. Fragen bleiben aber offen und werden solange wiederholt, bis sie beantwortet sind. Die ihnen zugrunde liegenden Anfragen des Benutzers werden dann ebenfalls beantwortet.

Diese Beispiele zeigen, daß bei Dialogsprachen eindeutig eine Tendenz zur Verringerung der Prozeduralität besteht. Ein weiterer Vorteil von Dialogsystemen ist die Möglichkeit, den Menschen dann in einen Lösungsprozeß einzuschalten, wenn Algorithmen überfordert sind, wenn die gewünschten Leistungen von Algorithmen nur mit unverhältnismäßig hohem Aufwand erbracht werden oder wenn der Benutzer sich Teilbeiträge zur Lösung in jedem Fall vorbehält. Beispiele dafür sind die speziellen graphischen Systeme EPIK ([HeVo 75]) und GRAPHBILD ([Zi 77]). Bei der interaktiven Kartographie ist die Extraktion von Höhenlinien oder sonstigen Linien aus topographischen Karten eine Teilaufgabe. Im lokalen Bereich gibt es dafür leistungsfähige Algorithmen. Sind die Originallinien jedoch gestört durch den Kartenrand, überlagerte Information o.ä., so lassen sich Korrekturen von Hand viel effektiver durchführen. Das andere Beispiel befaßt sich mit der ebenen Darstellung von Graphen. Die möglichst überschneidungsfreie und gefällige Darstellung aller Knoten und Kanten kann von Algorithmen zwar gut approximiert werden, die endgültige Feinarbeit bleibt aber besser dem Benutzer überlassen.

Im Dialogsystem EBIDIS ist der direkte Eingriff in den Entstehungsprozeß eines Bildes nach Sichtkontrolle möglich. Bildkorrekturen von Hand werden schon im Bildprogrammiersystem EHBIS [Br 76] durch einen lokalen, graphischen Dialog vorgenommen. Aufgabe von EBIDIS ist nun die Erstellung, Korrektur, Verknüpfung und Verwaltung von zweidimensionalen Bildern durch eine

Dialogsprache. Die in einem GMD-Bericht ([Dzi 77]) ermittelten Eigenschaften von benutzerfreundlichen Dialogsystemen sind im Dialogsystem EBIDIS weitgehend erfüllt.

4.2 Allgemeine Eigenschaften des Dialogsystems EBIDIS

Das Erlanger Bild-Dialogsystem EBIDIS ist ein Einzelbenutzer-System, das die quasiparallele Führung mehrerer Dialoge erlaubt. Es wird in der Umgebung des Einzelbenutzer-Betriebssystems DOS auf einer PDP-15 implementiert. Während eines Dialoges leistet EBIDIS Vorder- und Hintergrundarbeit: Im Vordergrund die Verwaltungsarbeit und die Ausführung von Benutzeraufträgen, im Hintergrund ständige Datensicherung und Übersetzungsarbeit "auf Verdacht". Zwei Übersetzer für verschiedene Sprachanteile (s. Abschnitt 4.3.2.3) erzeugen ausführbaren Code; da aber auch während der Ausführung von Aufträgen nachübersetzt werden kann, ist die Grenze zur interpretativen Bearbeitung verwischt. EBIDIS ist ein interaktives Dialogsystem: Die Eingabesprache wird zustandsabhängig eingeschränkt (s. Abschnitt 4.3.2.2), der Benutzer kann die Vorder- und Hintergrundarbeit unterbrechen. Eine Darstellung und gegenseitige Abgrenzung der Begriffe *Dialogsystem*, *Interaktionssystem* und *Kommunikationssystem* findet man in [KuWi 75]. Figur 5 zeigt die Struktur eines Interaktionssystems.

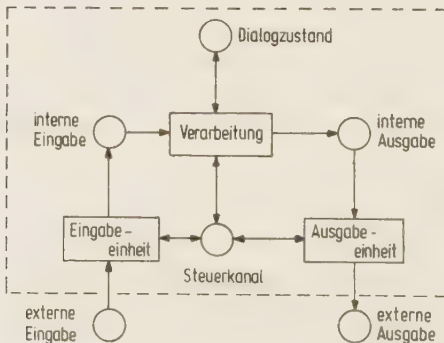


Fig. 5: Interaktionssystem

Dialogsysteme werden nach [KuWi 75] grob unterteilt in einen Rahmen und in einen Kern. Diese Unterteilung ist für den Benutzer weder wichtig noch sichtbar. Sie ist für die Implementierung insofern relevant, als ein Dialograhmen einen erheblichen Entwicklungsaufwand darstellt und der Dialogkern darum austauschbar sein muß. Es ist dann dem Benutzer möglich, verschiedene Problemklassen mit verschiedenen Dialogsystemen zu bearbeiten, aber doch dabei im wesentlichen die gleiche Dialogsprache zu verwenden. Eine grobe Aufteilung des Dialogsystems für EBIDIS zeigt Figur 6.

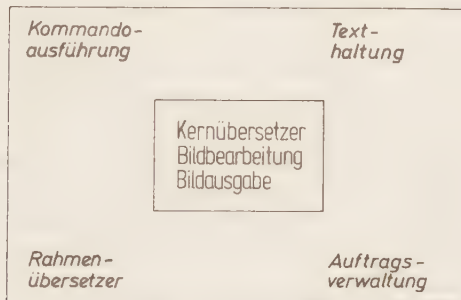


Fig. 6: EBIDIS Dialograhmen und -kern

Im einzelnen umfaßt der Dialograhmen die

Datenbereiche

und

Aufgaben

- Textspeicher für hinterlegte Anweisungen,
- Dialogauswahl und -wechsel,
- lexikalische Analyse der eingegebenen Zeilen,
- Kommandoentschlüsselung und -ausführung,
- Textverwaltung,

- | | |
|-------------------|---------------------------------------|
| - Auftragskeller, | - Verwaltung der Namensliste, |
| - Codespeicher, | - Auftragsverwaltung und -abwicklung, |
| | - Übersetzen der Rahmensprache, |
| | - Ein- und Ausgabe. |

Ein Dialogkern enthält problemorientierte Datenstrukturen und zugehörige Operationen. Sie sind nicht dialogspezifisch und treten ebenfalls in Programmiersprachen auf. Bei EBIDIS verursacht die graphische Ausgabe noch zusätzlichen Aufwand.

Der Dialogkern enthält die

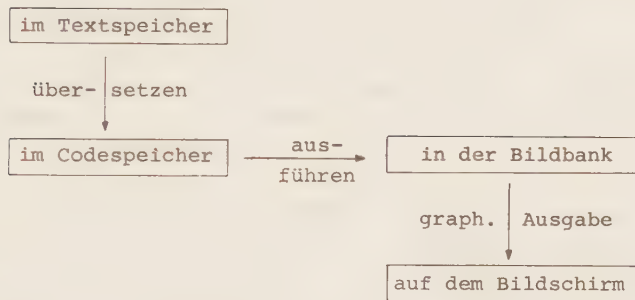
<u>Datenbereiche</u>	und	<u>Aufgaben</u>
- Namensliste (Variable),		- Übersetzen der Kernsprache
- Bildbank (strukt. Bilddaten),		- Operationen auf Bildern,
		- Generierung von Bildcode,
		- Bildausgabe über Bildschirm.

Bei den obigen Übersichten treten die Begriffe *Kommando*, *Rahmensprache* und *Kernsprache* auf. Sie konstituieren die Eingaben der Dialogsprache. Diese Sprachanteile werden nur vom Dialogsystem unterschieden. Vor dem Benutzer werden sie durch einheitlichen Sprachentwurf bewußt verborgen.

Wie entstehen nun Bilder im System EBIDIS? Man geht von einer Bezeichnung für ein geplantes Bild aus und weist ihr einen Ausdruck zu, der die Grobstruktur des Bildes beschreibt. Die darin enthaltenen Teilbilder werden anschließend beschrieben, und dieser Prozeß wird fortgesetzt, bis er bei primitiven Bildern oder bei vordefinierten Bildern endet. Dieses Vorgehen entspricht genau dem top-down-Entwurf von Algorithmen. Das Ergebnis sind ausführbare Anweisungen, die mit Bildausdrücken ([Sh 69]) das gewünschte Bild beschreiben. Werden diese Anweisungen ausgeführt, so bauen sie in der Bildbank die zugehörige Datenstruktur auf ([Br 76]). Von dort kann das Bild durch eine Ausgabeanweisung auf den Bildschirm gebracht und geprüft werden (Figur 7).

ANWEISUNGEN:

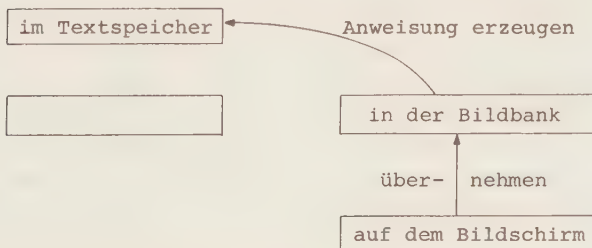
BILDER:

Fig. 7: Bildentwurf *top-down*

Der umgekehrte Prozeß ist in EBIDIS auch möglich. Am Bildschirm werden Teilbilder aus kleineren Einheiten versuchsweise zusammengesetzt zu immer komplexeren Bildern. Das geschieht unter Sichtkontrolle durch Angabe meist zweier Bildnamen und direkte Eingabe von Bildpositionen durch Steuerknüppel. Die Verbindung der Teile wird durch Angabe eines geeigneten verknüpfenden Operators fixiert, von denen EHBIS vier verschiedene kennt ([Br 76]). Diese gleichsam in bottom-up-Richtung entstandenen Bilder können in die Bilddbank übernommen werden und auch in den Textspeicher. Dazu werden den Konstruktionsprozeß begleitend ausführbare Anweisungen erzeugt, die auch genau das konstruierte Bild erzeugt hätten (Figur 8).

ANWEISUNGEN:

BILDER:

Fig. 8: Bildentwurf *bottom-up*

Beide Entwurfsmethoden werden in EBIDIS verwirklicht und können abwechselnd eingesetzt werden. Es stehen dafür die Kommandos NEUER TEXT bzw. NEUES BILD zur Verfügung (Abschnitte 4.3.1.2 und 4.3.1.3). Dieses Vorgehen entspricht genau der in Abschnitt 3.3.3 beschriebenen "Yo-yo-Programmierung" von Algorithmen. EBIDIS unterscheidet sich darin von anderen Entwicklungen, die nur für eine Entwurfsrichtung ausgelegt wurden. Beispiele dafür sind PAD ([Mu 74]) und ELAN ([HeJä 77]).

Bestehen die Systemäußerungen sowohl aus Textzeilen wie auch aus graphischer Information, so kann man dafür zwei verschiedene Sichtgeräte benutzen (wie z. B. beim Dialogsystem AMTRAN). EBIDIS wickelt den Textdialog wahlweise über eine Schreibmaschine oder das Sichtgerät ab. Im letzteren Fall erlaubt eine Funktionstaste (alt mode) das beliebige Wechseln zwischen graphischer und alphanumerischer Ausgabe am Bildschirm.

4.3 Bildbearbeitung mit EBIDIS

Ziel eines Bilddialoges in EBIDIS ist die Konstruktion, Bearbeitung und Verwaltung zweidimensionaler Bilder. Der grobe Ablauf der Entstehung von Bildern in beiden Entwurfsrichtungen wurde im vorigen Abschnitt beschrieben. Hier sollen die einzelnen Tätigkeiten im Dialog genauer vorgestellt werden. Sie werden in Abschnitt 4.3.1 zunächst inhaltlich beschrieben in ihrer Wirkung. Mit der Formulierung von Tätigkeiten in Kommandos beschäftigt sich später Abschnitt 4.3.2.3.1.

4.3.1 Tätigkeiten bei der Verwaltung von Objekten

Alle Tätigkeiten beziehen sich auf Objekte, und zunächst sollen diese Objekte des Dialogsystems vorgestellt werden. Die von Programmiersprachen her bekannten Variablen gibt es auch in Dialogsystemen. Sie werden als *Namen* bezeichnet und dem Dialogkern zugerechnet. Besondere Variable sind die *Bilder*. Hinterlegte Textzeilen werden als *Teile* bezeichnet. Gelangen sie zur Ausführung, so heißen sie *Aufträge*. Beide Objektarten werden dem Dialograhmen zugerechnet. Ein *Dialog* mit seinen

Objekten ist selbst ein Objekt, das von EBIDIS verwaltet wird. Diese Eigenschaft von Dialogsystemen wird als *Reflexivität* bezeichnet. Im Dialog kann über den Dialog und seine Bestandteile gesprochen werden.

Die Tätigkeiten nun müssen vom Benutzer für alle Objektarten als ähnlich empfunden werden; darum können sie in EBIDIS auch einheitlich beschrieben werden. Sie werden bezeichnet mit *neu*, *zeige*, *lösche*, *speichere*, *hole* und *komprimiere*. Objektarten und Tätigkeitsbezeichnungen bilden eine orthogonale Basis, in der einzelne Tätigkeiten eingeordnet werden. Fast alle möglichen Kombinationen von Bezeichnungen sind sinnvolle Tätigkeiten und in EBIDIS definiert, viele davon werden durch die Dialogsprache explizit ausgedrückt, manche Tätigkeiten werden implizit vom Dialogsystem erbracht.

4.3.1.1 Namensliste

Die Objekte des Dialogkerns sind vom Typ *integer*, *picture* und *reserviert*. Ihre Bezeichner beginnen mit einem Buchstaben, enthalten beliebig viele Buchstaben oder Ziffern und keine Zwischenräume oder Sonderzeichen. Der Benutzer kann der Namensliste neue Namen vom Typ *integer* oder *picture* hinzufügen oder eindimensionale Felder solcher Namen. Ein entsprechendes Kommando lautet z. B.

NEUE INTEGER NAMEN A, B(50), C

Kommandos werden sofort ausgeführt. Sie können vorerst nicht hinterlegt werden im Gegensatz zu Anweisungen. Diese Einschränkung geht aber nur auf die Kernspeicherkapazität zurück. Wenn spätere Arbeiten am Dialogsystem erlauben, den Code der Kommandoausführung und der Auftragsabwicklung evtl. gleichzeitig in den Kernspeicher zu laden, dann können auch Kommandos erst gespeichert werden und dabei sogar zu Kommandoprozeduren (vgl. Abschnitt 4.3.1.3) zusammengefaßt werden.

Der Typ *picture* nimmt strukturierte Bilddaten auf, die Wortsymbole in den Eingaben der Dialogsprache sind reserviert. An

weiteren Typen sind vorgesehen *real*, *Boolean*, *picproc* und *position*. Vom Typ *picproc* sind Bilder, die nicht in der Bildbank zur Bearbeitung vorliegen, sondern als ausgabefertiger Bildcode von anderen Bildern aufgerufen werden. Größen vom Typ *position* markieren Stellen in einem Bild, die bildbezogen definiert werden, z. B. den Anfangspunkt für das Zeichnen eines Teilbildes. Das erlaubt bei einer späteren Erweiterung der Kernsprache das bequeme Beschreiben von Querverbindungen in einem Bild.

Namen können der Namensliste auch implizit zugeführt werden. Ihr Typ wird dann vom Kernübersetzer aus dem Kontext ermittelt und nach Erlaubnis des Benutzers festgelegt.

Die Inspektion der Namensliste geschieht durch die Tätigkeit *zeige*. Genauere Angaben werden erfragt oder stehen gleich im Kommando, wie etwa in

ZEIGE ALLE NAMEN

ALLE INTEGER NAMEN ZEIGEN

ZEIGE NAMEN A, B, G, H

Hierbei wird noch zwischen einer Inspektion "en gros" unterschieden (z. B. "alle Namen", "alle integer Namen") und einer Inspektion "en detail". Ist das Feld B(50) z. B. vom Typ *integer*, so wird es beim zweiten Kommando nur mit seinem Typ, der Feldgröße und einer Angabe ausgedruckt, ob überhaupt Elemente darin definiert sind. Beim dritten Kommando werden dagegen alle Feldelemente einzeln aufgelistet. Genauso verhält es sich bei Größen vom Typ *picture*, also zusammengesetzten Daten. Einmal wird nur ein Verweis ausgegeben, der in die Bildbank zeigt, das anderemal alle Information aus der Bildbank dazu. Handelt es sich sogar um ein Feld vom Typ *picture*, das etwa mit dem Kommando

NEUER PIC NAME P(20)

eingeführt wurde, so bewirkt das Kommando

ZEIGE NAMEN P

die Ausgabe aller Feldelemente mit Verweisen und erst das Kommando

ZEIGE NAMEN P(15)

etwa die Information in der Bildbank zu einem Bild.

Beim Streichen von Namen aus der Namensliste muß jeder Name einzeln angegeben werden, z. B.

LOESCHE NAMEN B, G, H

Zum Schutz vor ungewolltem Löschen sind Kommandos mit der Angabe "lösche alle ..." hier nicht erlaubt. Der dabei freiwerdende Platz in der Namensliste kann erst nach dem Kommando

KOMPRIMIERE NAMEN

wieder verwendet werden. Eine implizite Speicherbereinigung bei Überlauf der Namensliste ist in EBIDIS grundsätzlich möglich, wird aber wegen einer Nebenwirkung nicht ausgelöst: Verändern sich die relativen Adressen von Namen in der Namensliste, so ist der bisher erzeugte Code für Anweisungen nicht mehr gültig und der ganze Codespeicher wird gelöscht. Ähnliches gilt bei der Bereinigung der anderen Speicherbereiche für die Bildbank und für die Anweisungstexte,

Für die Verwendung von Hintergrundspeicher zum Zwischenspeichern oder Ablegen von Information sind die Tätigkeiten *speichere* und *hole* vorhanden. Damit können einzelne Namen ausgelagert werden, und hier ist besonders an Objekte mit großem Speicherbedarf gedacht wie die vom Typ *picture* oder *picproc*. Dies geschieht entweder zum Entlasten und Vergrößern der Bildbank, die zu einem Dialog gehört, oder zur Übergabe von Bilddaten aus einem Dialog zu einem anderen. Die weitere Bearbeitung von Bildern erfolgt entweder noch innerhalb von EBIDIS durch zusätzliche Operationen - denkbar ist etwa die Berechnung des Inhalts von umschlossenen Flächen - , die dann in die Dialogsprache mit aufgenommen werden, oder durch unabhängige Programmsysteme. Dazu

werden Bilder als Datenstrukturen (Typ *picture*) auf Hintergrundspeicher abgelegt und von dort zur unabhängigen Weiterbearbeitung abgeholt. Objekte vom Typ *picture* enthalten noch alle Information zur Bildstruktur, die dem Dialogsystem bekannt war. Zusätzlich können über Bildattribute den Bildern oder Teilen davon noch Texte und numerische Größen mitgegeben werden (z. B. als Bezeichnung eines Anlagenteils oder als Kapazität einer Rohrleitung). Diese Attribute sind nicht sichtbare Bildanteile, sondern weitere Angaben für nachgeordnete Programmsysteme. Die ausgabefertigen Bilder, Objekte vom Typ *picproc*, werden als Binärdateien ausgegeben. Sie können vom Systemlader bei unabhängigen Programmen, etwa Lehrdialogen o. ä., dazugeladen werden und erlauben noch das dynamische Ändern enthaltener Textzeilen und den Einsatz eines Lichtgriffels.

Über Wertzuweisungen an die Objekte des Dialogkerns und die inhaltliche Bearbeitung der Bildbank wurde bisher noch nichts gesagt. Das geschieht bei der Behandlung von Anweisungen in EBIDIS. Die Variablen des Dialogkerns haben anfangs den Wert "undefiniert". Er verursacht beim Lesen eine Unterbrechung des entsprechenden Auftrages.

4.3.1.2 Bildbank

Die Bildbank gehört formal zur Namensliste und entstand aus der Werteliste für Objekte des Typs *picture*. So ergeben sich für zwei Tätigkeiten, nämlich *speichern* und *holen*, dieselben Wirkungen wie bei der Namensliste. Das Kommando

SPEICHERE BILD B2

etwa wirkt genauso wie "speichere Namen B2", wenn B2 vom Typ *picture* ist. Beim Zeigen von Bildern ergibt sich gegenüber dem Zeigen von Namen eine Einschränkung: Die graphische Ausgabe eines Bildes benötigt den ganzen Bildschirm. Mehrere einzelne Bilder müssen nacheinander ausgegeben werden. Der Übergang von einem zum nächsten Bild erfolgt dann durch Tastendruck oder nach einer im Kommando angegebenen Zeit:

BILDER B4, B5, BI(15) ZEIGEN

ZEIGE BILD ABC 5 SEC, DEH 10 SEC

ZEIGE ALLE BILDER

Das letzte Kommando listet alle Bildnamen mit definiertem Wert auf. Das Kommando

LOESCHE BILDER BI(10), B4, B2

löscht nur die Bilddaten in der Bildbank. Die Namen in der Namensliste bekommen den Wert "undefiniert" und bleiben dort erhalten. Eine Speicherbereinigung in der Bildbank durch

KOMPRIMIERE BILDER

wird wie bei der Namensliste vom Benutzer ausgelöst. Die Eingabe

NEUES BILD

eröffnet nun den in Abschnitt 4.2 beschriebenen Bildaufbau in der Richtung *bottom-up*. Es folgen Benutzerangaben über einen oder zwei Bildoperanden. Dabei handelt es sich um primitive Bilder, einfachste Bildelemente wie Strecken, Kreisbögen o. ä., oder um bereits vorhandene Bilder, auf denen weiter aufgebaut wird. Diese Operanden werden zur Kontrolle abgebildet und dann gekellert. Nach Angabe eines der in [Br 76] dargestellten Bildoperatoren erscheint das neu zusammengefügte Bild auf dem Bildschirm. Es kann dann korrigiert werden durch Vertauschen oder Verändern der Operanden und anschließend abgelegt werden. Entweder wird es als neuer Bildoperand gekellert oder unter einem Bildnamen in der Bildbank abgelegt. Die entsprechend der Konstruktion erzeugte Anweisung kann unter einem Text-Bezeichner im Textspeicher abgelegt werden.

4.3.1.3 Textspeicher

Der Textspeicher nimmt Anweisungen zur späteren Ausführung auf. Der hinterlegte Text aller Anweisungen ist hierarchisch gegliedert in sogenannte "Teile". Sie entsprechen den in der Dialogsprache JOSS (s. [KuWi 75]) verwendeten ausführbaren Einheiten, den *parts* und *steps*. Bei EBIDIS können die ausführbaren Einheiten jedoch über mehrere Stufen verfeinert werden. Teile werden unter numerischen Bezeichnungen abgelegt, die auch bei der Unterteilung von Kapiteln und Abschnitten verwendet werden. Ein Teil besteht aus wenigstens einer Textzeile, die unter einer Bezeichnung, z.B. "2.3", abgelegt ist, und vielleicht aus einer Anzahl von Unterteilen, deren Bezeichnung entsprechend verlängert ist, also etwa zu "2.3.10" oder "2.3.7". Bei der Eingabe von Teilen wird dem Benutzer vom Dialogsystem nach Möglichkeit eine Bezeichnung vorgeschlagen. Z.B. wird nach Eingabe der Zeile 2.3.5 die Teilbezeichnung

TEIL 2.3.6 ?

angeboten, die der Benutzer akzeptieren oder verwerfen kann. Die aufsteigend sequentielle Eingabe von Teilen wird dadurch erleichtert, aber nicht vorgeschrieben. Reihenfolge und Anzahl der eingegebenen Unterteile ist beliebig, allerdings muß die jeweils unmittelbar übergeordnete Textzeile dann schon vorhanden sein. So sind Reihenfolge und Lücken unter den Teilbezeichnungen bei der Eingabe von Zeilen in horizontalen Ebenen frei wählbar, in vertikalen dagegen nicht.

Bei der Implementierung des Textspeichers wurde die Anzahl der vertikalen Unterteilungen auf sechs beschränkt. Das bedeutet, daß jede Teilbezeichnung höchstens sechs Zahlen enthält, für jede Unterteilungsstufe eine. Ebenso wurde die Anzahl von Unterteilen für jeden Teil auf 15 beschränkt. Diese Grenzwerte sind jedoch leicht zu vergrößern. Der Inhalt des ganzen Textspeichers kann mit der Bezeichnung "alle Teile" angesprochen werden.

Diese Textstruktur ist den Erfordernissen angepaßt, wie sie beim top-down-Entwurf durch schrittweises Verfeinern vorliegen. Die Spitze des Entwurfsbaumes entspricht hier dem Bildnamen für ein zu entwerfendes Bild, z.B. dem Namen ANSICHT:

```
.  
.  
TEIL  2.3  
                                ANSICHT = VORNE + SEITE  
TEIL  2.3.2  
                                VORNE = .....  
TEIL  2.3.3  
                                SEITE = .....  
TEIL  2.3.2.1  
      .  
      .  
      .
```

Das schrittweise Verfeinern entspricht hier der schrittweisen Dekomposition der jeweiligen Teilbilder entsprechend ihrer Bildstruktur. Das geschieht durch Formulierung der Bildanweisungen. Die Synthese des so beschriebenen Bildes erfolgt dann bei Ausführung des Teils 2.3. Hierbei werden die Anweisungen in den untersten Teilen zuerst abgearbeitet. Die Anweisung in der Zeile eines Teils wird erst nach Fertigstellung aller Unterteile ausgeführt, diese Unterteile werden in aufsteigender Reihenfolge behandelt. Mit der Ausführung der Zeile 2.3 ist dieser Auftrag dann beendet.

Die Ausführung aller Anweisungen eines Teiles erfolgt also prozedural. Insbesondere stellt jeder Teil eine zunächst parameterlose Prozedur dar, die durch Anweisungen wie etwa

```
TEIL  2.7.10  
      RECHNE TEIL 2.3
```

ausgeführt wird. Durch Zusätze bei einer RECHNE-Anweisung wie

```
RECHNE TEIL  2.3 MIT A = 3
```

werden globale Größen zuvor gesetzt, und dadurch wird sogar

ein einfaches Parameterkonzept realisiert. Es gibt in EBIDIS keine Blockstruktur. Alle Teile sind in demselben globalen Block deklariert, der durch den Dialog dargestellt wird und der auch die Objekte des Kerns enthält. Der rekursive Aufruf von Teilen ist erlaubt.

Zur Verwaltung des Textspeichers stehen wieder die eingangs erwähnten sechs Tätigkeiten zur Verfügung:

NEUER TEXT

ZEIGE TEXT

.

KOMPRIMIERE TEXT

Bezieht sich eine Tätigkeit auf den Inhalt des ganzen Textspeichers (außer *neu*), so kann es im Kommando gleich angegeben werden, wie z.B. in

ZEIGE ALLEN TEXT

Sonst wird ein Bezeichnung erfragt oder vorgeschlagen (außer *komprimiere*). Die von einem Dialogsystem geforderte Transparenz beinhaltet auch ein leichtes Verstehen und Überprüfen der vom System erbrachten Leistungen durch den Benutzer. Die hier beschriebenen Tätigkeiten sind solche Leistungen. Beim Zeigen von Text erübrigt sich eine Überprüfung. Nur in dem Fall, daß der Textspeicher den gewünschten Textteil nicht enthält oder ganz leer ist, wäre eine leere Ausgabe als Quittung verwirrend. Darum erfolgt dann eine entsprechende Meldung. Beim speichern und holen der Teile vom Hintergrundspeicher werden alle transportierten Zeilen protokolliert. Die Ausführung von *speichere*, *hole* oder *lösche* wird bestätigt.

Manche Aktionen dürfen nicht in jedem Falle ausgeführt werden. Durch Rückfragen des Dialogsystems vor und nach der eigentlichen Tätigkeit wird die vom Benutzer gewünschte Wirkung erzielt. Bei den Tätigkeiten *neu*, *holen* und *speichern* kann der bezeichnete Teil an der Stelle des Transportzieles schon vor-

handen sein. Dann muß der Benutzer die Ersetzung ausdrücklich erlauben. Nach dem Transport kann die Information an der Quelle überflüssig werden. Sie wird dort nach ausdrücklicher Aufforderung gelöscht. Ein Beispiel für einen solchen Dialogabschnitt folgt (die Benutzereingaben sind unterstrichen):

```
.  
.
ALLEN TEXT SPEICHERN
AUF HINTERGRUNDSPEICHER ERSETZEN ? JA

2.3          < Anweisung >
2.3.1        < Anweisung >
2.3.2        < Anweisung >
              . . . . .
2.3.2.2      . . . . .
.
.
2.3.4.5      . . . . .

GESPEICHERT

AUCH LÖSCHEN ? JA
GELÖSCHT
.
.
```

Gelöscht wurden die angegebenen Zeilen im Textspeicher. Bei der Eingabe neuer Zeilen sind im Systemhintergrund die beiden Übersetzer tätig. Eine vollständig eingebene Anweisung wird erst dem Kernübersetzer angeboten. War es keine Anweisung der Kernsprache, so wird der Rahmenübersetzer aktiviert. Das geschieht in den Systemwartezeiten während der Eingabe der nächsten Zeile.

4.3.1.4 Auftragskeller

Die bisher besprochenen Benutzeräußerungen waren Kommandos, die im Grundzustand des Dialogsystems nach der Aufforderung

GIB KOMMANDO, ANWEISUNG ODER "?"

eingegeben wurden. Anweisungen gehören zur Kern- oder Bildsprache (z.B. ANSICHT = VORNE + SEITE) oder zur Rahmensprache (z.B. RECHNE TEIL 2.3). Sie erzeugen und starten unmittelbar einen Auftrag, wenn sie in obigem Grundzustand eingegeben werden. Ein Auftrag, der wegen einer Unterbrechung (z.B. Teil nicht vorhanden, Name nicht vorhanden, Wert nicht definiert) nicht ganz ausgeführt werden kann, bleibt im Auftragskeller, und das Dialogsystem geht wieder in den Grundzustand "Direkte Eingabe" über. So ist ein Auftrag die dynamische Erscheinungsform einer statischen Anweisung, die diesen Auftrag auslöst. Diese Anweisung wird im Auftragskeller mit abgelegt für die Zwecke der Kellerinspektion und des Nachübersetzens.

Bei der Bearbeitung des Auftragskellers fehlen die Kommandos für die beiden Tätigkeiten *speichere* und *hole*. Sie werden implizit mit ausgeführt, wenn der Benutzer den aktuellen Dialog beendet und nicht löscht, um ihn bei einer späteren Sitzung weiterzuführen. Die Inspektion des Auftragskellers ist etwa durch das Kommando

ZEIGE AUFTRÄGE

möglich. Dann werden immer alle gekellerten Aufträge und der eine zuletzt unterbrochene Auftrag mit ihren auslösenden Anweisungen aufgelistet. Es ist nur das Löschen des jeweils zuletzt unterbrochenen Auftrages oder aller Aufträge möglich z.B. durch die Kommandos

ALLE AUFTRÄGE LÖSCHEN oder

LÖSCHE AUFTRAG

So ist eine Speicherbereinigung im Auftragskeller nicht erforderlich. Eine Sonderstellung nimmt hier die Tätigkeit "neuer Auftrag" ein. Diese Tätigkeit wird ja implizit durch direkte Eingabe einer Anweisung ausgelöst. Es bleibt nur, die Fortsetzung des jeweils zuletzt unterbrochenen Auftrages durch ein Kommando anzustoßen, nämlich durch

WEITER

Das Dialogsystem wickelt dann den zuletzt unterbrochenen Auftrag weiter ab. Kommt es zu seinem Ende, so werden die zuvor gekellerten Aufträge der Reihe nach verarbeitet, bis eine neue Störung die Bearbeitung eventuell wieder unterbricht. Sonst meldet das System FERTIG. Bei manchen Unterbrechungen möchte der Benutzer auf die geforderte Korrektur verzichten. Ein Teil ist nicht vorhanden, ein Wert ist nicht definiert, aber der Benutzer erkennt, daß dieser Fehler für den weiteren Ablauf nicht wichtig ist. Dann wird die Fortsetzung des Auftrages über einen solchen Fehler hinweg durch das Kommando

WEITER !

erzwungen.

4.3.1.5 Dialogverwaltung

Bei der Verwaltung von Dialogen hat es der Benutzer besonders leicht. Das hat den arbeitspsychologischen Grund, daß der Beginn einer Sitzung oder der Übergang von einem Dialog zu einem anderen mehr vom Dialogsystem geführt sein soll, als etwa vom Benutzer erfragt oder gar erwartet. So beginnt jede EBI-DIS-Sitzung mit dem selbständigen "Zeigen aller Dialoge". Von früheren Sitzungen erhaltene Dialoge werden vom Hintergrundspeicher aufgelistet und mit einer Nummer und ihrer Dialogbezeichnung identifiziert. Der Benutzer wählt dann einen alten Dialog aus durch Eingabe seiner Nummer oder einen neuen durch Eingabe einer alphanumerischen Bezeichnung. Ein alter Dialog wird vom Hintergrundspeicher geholt mit all seinen Datenberei-

chen und fortgesetzt, ein neuer unmittelbar begonnen. Bei Dialogende kann der Benutzer diesen Dialog löschen, sonst wird er im aktuellen Zustand gespeichert. Wählt der Benutzer nun keinen Dialog mehr aus, so beendet EBIDIS die Sitzung.

Die Theorie der formalen Dialogsysteme ([KuWi 75]) kennt den Begriff eines *protokollführenden Dialogsystems*, das im Dialogzustand jederzeit die gesamte bisherige Eingabefolge widerspiegelt. Ein reales Dialogsystem kann dieser Definition dadurch genügen, daß es alle Benutzereingaben begleitend in eine Systemdatei kopiert, deren Inhalt dann zum Dialogzustand dazugenommen wird. Es ist bei EBIDIS später vorgesehen, daß etwa nach dem Kommando

SPEICHERE ALLE EINGABEN

diese Systemdatei angelegt und beschrieben wird. So eine Art der Dialogspeicherung faßt den Dialog als Tätigkeit auf, die jederzeit reproduziert werden kann. Einen Dialog "in der Konserve" kann man dann nachspielen, indem man einen aktuellen Dialog eröffnet und durch die Kommandos

HOLE EINGABE oder

SPEICHERE ALLE EINGABEN

den konservierten Dialog in einzelnen Schritten oder fortlaufend nachvollzieht. Durch rechtzeitiges Unterbrechen kann man so alle Zustände des konservierten Dialoges erreichen. EBIDIS speichert vorläufig aber nur die aktuellen Ergebnisse eines Dialoges, die in seinen Datenbereichen verfügbar sind.

Da auch die Systemäußerungen zu einem Protokoll gehören, ist es denkbar, mit den Benutzereingaben auch alle Systemausgaben, veranlaßt etwa durch das Kommando

SPEICHERE ALLE AUSGABEN

mit in die Systemdatei zu kopieren. Dieses Dialogprotokoll

kann dann allerdings nicht mehr nachgespielt werden, sondern nur noch von Hand ausgewertet werden.

4.3.2 Die Dialogsprache

4.3.2.1 Formale Betrachtung

Sprache dient der Formulierung und Fixierung von Gedanken. Umgekehrt werden Denkprozesse durch sinnvolle Sätze ausgelöst, dirigiert und korrigiert. Die Frage ist dann, wer dominiert über wen? Da die Gesamtheit gedankenloser Sätze wohl vernachlässigt werden kann gegenüber der Gesamtheit nicht formulierter Gedanken, darf man die Gesamtheit sprachlicher Information als eingebettet betrachten in der Gesamtheit aller Gedanken. Dieser Sachverhalt kann mit Begriffen der Linearen Algebra etwas konkreter beschrieben werden. Ein Vektorraum aller Gedanken wird von Unterräumen verschiedener Dimensionen, den Sprachen, durchzogen. Denkprozesse sind wie Kurven im Gesamtraum, die auf Unterräume projiziert oder mit solchen geschnitten werden. Überträgt man diesen Vergleich auf die Situation eines Dialoges mit einem Rechen-

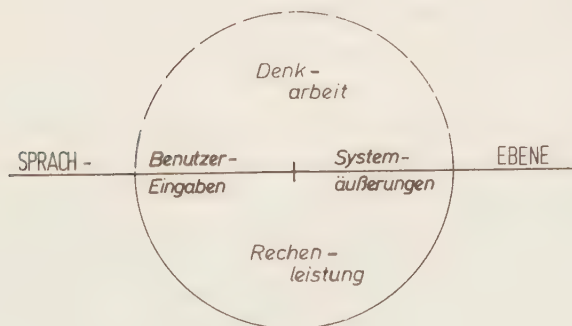


Fig. 9: Sprache als Hyperebene

system, so erhält man zwei Halbräume und eine trennende Hyperfläche (Figur 9). Der Bereich der Rechnerleistung kann eingegrenzt werden, der Bereich der Denkarbeit nicht. Zwischen beiden liegt die Sprachebene. Ein Dialog ist nun wie eine Kurve, die abwechselnd durch beide Halbräume führt und dabei die Sprachebene auch abwechselnd auf der Seite der Benutzereingaben und der Systemausgaben schneidet (Figur 10). Diese Kurve verläuft sonst frei.

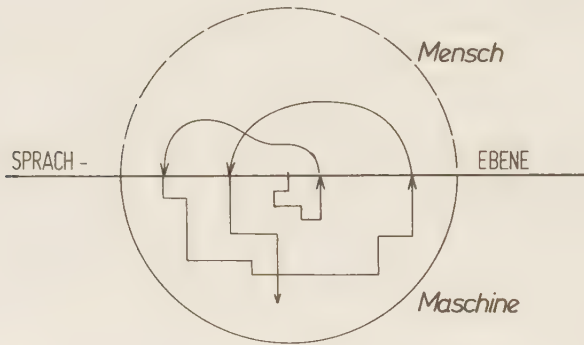


Fig. 10: Dialog und Dialogprotokoll

Kleine Teilwege durch den Denkbereich entsprechen etwa der Beantwortung einer System-Rückfrage, große Teilwege durch den Rechenbereich etwa der Abwicklung eines Auftrages. Was bei diesem Dialogmodell fehlt, ist die Darstellung der Zustände, die ein Dialogsystem während eines Dialoges annimmt, und die Tatsache, daß ein Dialog zielgerichtet ist. Das liegt daran, daß die Halbräume des Modells nicht näher untersucht und strukturiert werden. Was dieses Modell aber deutlich zeigt, ist die Einbettung eines Dialogprotokolls in den eigentlichen Dialog. Das Dialogprotokoll ist nur

ein zeitlich angeordnetes Schnittbild von viel komplexeren Vorgängen mit der Sprachebene.

Dieses Schnittbild ist nichts anderes als der bei [KuWi 75] definierte formale Dialog

$$a_1 b_1 a_2 b_2 \dots a_n b_n, \quad n \geq 1$$

mit $a_i \in \text{Eingabesprache}$

und $b_i \in \text{Ausgabesprache}$.

Vergleicht man einen formalen Dialog mit einem herkömmlichen Programmiersprachen-Programm, so erkennt man einen auffallenden Unterschied: Beim Dialog sind die Systemäußerungen Teil der Sprache, beim Stapelbetrieb nicht. Die Rückmeldungen des Übersetzers und des ausführenden Laufzeitsystems entstehen an verschiedenen Stellen des Betriebssystems und teilweise unabhängig von der verwendeten Programmiersprache, beim Dialog bedienen sich beide Partner derselben Sprache. Ein Beispiel dafür sind die Bildsprachenanweisungen, die EBIDIS bei der bottom-up-Konstruktion von Bildern erzeugt (Abschnitt 4.2) und die auch vom Benutzer eingegeben werden (Abschnitt 4.3.1.3, 4.3.2.3.2).

Ein wichtiger Aspekt ist der Einfluß des Dialogzustandes auf die Sprache. Formal wird jedes b_i im formalen Dialog als abhängig betrachtet von der Folge aller vorausgegangenen Eingaben:

$$b_i = v(a_1 a_2 \dots a_i) \quad \text{für } i = 1, \dots, n$$

Die Funktion v , die jeder Folge von Eingabeworten die zuletzt erhaltene Ausgabe zuordnet, bestimmt das Verhalten des formalen Dialogsystems und damit auch die Sprache. Stellt man jedoch das formale Dialogsystem als einen unendlichen, initialen Mealy-Automaten dar ([KuWi 75]), so kann man die Eingabefolge $a_1 a_2 \dots a_{i-1}$ abbilden auf den Automatenzustand s_{i-1} und b_i vermöge der Ausgabefunktion l definieren als

$$b_i = l(s_{i-1}, a_i) \quad (i > 0)$$

Der Teilweg im Dialogmodell von einer Eingabe a_i durch den Bereich der Rechenleistung zur Ausgabe b_i entspricht dann gerade dem Zustandsübergang von s_{i-1} nach s_i . Dieser Übergang erfolgt deterministisch. Der Benutzer orientiert sich am Dialogzustand, der ihm in der Regel durch die letzte Ausgabe verdeutlicht wird, und überlegt sich die nächste Eingabe. Dieser Übergang ist nichtdeterministisch und ermöglicht die vielfältige Dialogfortsetzung.

4.3.2.2 Eigenschaften realer Dialogsprachen

Für eine reale Dialogsprache ergeben sich daraus verschiedene Eigenschaften. Zunächst aber sei festgestellt, daß die Benutzereingaben und Systemäußerungen sich nicht auf die sprachliche Ebene beschränken. Hinzu kommen der akustische Bereich und der Einsatz verschiedener Geräte, die den Dialog auch tragen. Am Beispiel EBIDIS wird das in Abschnitt 4.3.2.3.4 beschrieben. Hier wird erst der sprachliche Anteil der Verständigung betrachtet, und dabei auch mehr die spontanen Äußerungen beider Partner und weniger die Informationsdarstellung in Tafeln und Listen.

4.3.2.2.1 Ein- und Ausgabesprache

Die Eingabesprache dient dem Benutzer zur Formulierung seiner Wünsche und Fragen. Benutzernähe der Eingaben wird durch Problemnähe und Ausdrucksnähe erreicht. Die Nähe zum Problem erklärt die Ähnlichkeit von existierenden Dialogsprachen zu problemorientierten Programmiersprachen. Die natürliche Ausdrucksweise des Benutzers kann von einem Dialogsystem nur näherungsweise verstanden und nachgebildet werden. Bei EBIDIS wird dazu von der herkömmlichen Syntaxanalyse teilweise abgewichen zugunsten einer inhaltlichen Freiwortanalyse.

Die verschiedenen Funktionen der Ausgabesprache, Information und Motivation des Benutzers, erfordern eine hohe Investition bei der Implementierung. Die Ausgaben sind Teil derselben

Dialogsprache. Der ausgegebene Klartext muß der Eingabesprache ähnlich sein. Von Abkürzungen ist unbedingt abzu-sehen. Günstig ist eine freie Kombinierbarkeit von Text-teilen, wobei jedoch Formulierungen "ohne Grate und Fugen" entstehen müssen. Diese freie Kombinierbarkeit entspricht der inhaltlichen Analyse von Eingaben. Ein intelligenter Partner soll sich auch klug und verständlich ausdrücken. Klug bedeutet, daß nicht nur der zurückliegende Rechenschritt nachgewiesen wird, sondern der erreichte Dialogzustand mit aller für das Kurzzeitgedächtnis wichtigen Information transparent gemacht wird. Ein verständlicher Ausdruck bemüht sich, die Wege des Verstehens beim menschlichen Partner zu erfüllen. Jede Systemäußerung ist ein Beitrag zum Dialog, und eine höfliche und hilfreiche Ausgabe ist für den Benutzer Motivation und Antrieb.

4.3.2.2.2 Der Einfluß des Dialogzustandes

Der Dialogzustand ist zu jeder Zeit durch das Dialogsystem definiert. Um seinen Einfluß auf eine Dialogsprache zu verdeutlichen, muß zunächst der Zustand selbst beleuchtet werden. Er läßt eine statische und eine dynamische Komponente erkennen. Als statische Komponente werden die in Speicherbereichen gesammelte Information und weitere Zustandsdaten betrachtet. Diese Zustandskomponente kann auf Hintergrundspeicher gesichert werden. Der Benutzer orientiert sich bezüglich des statischen Dialogzustandes durch eigene Kommandos, welche die Tätigkeit *Zeigen* enthalten. Diese Zustandskomponente bestimmt neben der dynamischen auch die Wirkung von Benutzereingaben auf das Dialogsystem. Durch die Eingabe

RECHNE TEIL 3.2.4

wird z. B. ein Auftrag erzeugt, wenn Teil 3.2.4 auch vorhanden ist, sonst eine Rückmeldung. So wirkt der statische Zustand auch auf die Systemausgaben.

Der dynamische Dialogzustand wird durch die momentan aktivierten Systemtätigkeiten beschrieben. Soweit er dem Benutzer bewußt sein muß, wird er ständig in den Systemäußerungen vergegenwärtigt. Dadurch wird die Benutzerführung bewirkt oder dem Benutzer die laufende Überwachung des Dialoges ermöglicht, je nach dem Umfang der Wahlmöglichkeiten bei der Eingabe. Beispiele für dynamische Dialogzustände im normalen Gesprächsverlauf sind der Systemzustand "Sammeln von Anweisungen" oder der Zustand "Ausführen von Anweisungen". Das Dialogsystem EBIDIS ist hierarchisch gegliedert. Beim Tiefersteigen in spezielle Tätigkeiten schränken sich auch die benötigten Benutzereingaben ein. Diese Einschränkung ist bei EBIDIS nicht als Verbot bestimmter Eingaben aufzufassen, sondern als eingeschränkte Auswertung der Benutzeräußerungen. Bei der Dialogsprache DIALOG (s. [KuWi 75]) wird dem Benutzer der jeweils aktuelle Zeichensatz für gültige Textfortsetzungen angezeigt. Andere Zeichen können nicht eingegeben werden. Bei EBIDIS werden zwar auch die aktuell möglichen Wortsymbole vorgeschlagen. Aber der geübte Benutzer kann teilweise mehr eingeben: Enthält seine Äußerung sinnvolle Angaben auch für die folgenden Systemfragen, so werden diese unterdrückt und der Dialog kürzt sich ab. So wirkt auch der dynamische Systemzustand auf die Benutzereingaben und Systemausgaben und damit auf die Dialogsprache.

Als dynamischer Zustand muß auch betrachtet werden, ob ein Übersetzer Anweisungen im Vordergrund des Dialogsystems während der Auftragsbearbeitung oder im Hintergrund während der Texteingabe bearbeitet. Bei Hintergrundarbeit werden Rückfragen und Fehlermeldungen unterdrückt. Wieder ist der Einfluß des Zustandes auf die Dialogsprache offenkundig.

4.3.2.2.3 Darstellung von Fehlersituationen

Die Verständlichkeit der Systemäußerungen bildet das Gegenstück zur Benutzernähe der Eingabesprache. Der in Abschnitt 4.3.1 schon beschriebenen Orthogonalität der Benutzereingaben steht die freie Kombinierbarkeit von Textteilen bei einigen Systemäußerungen gegenüber. Die Ausgabesprache hat die schwierige Aufgabe zu erfüllen, die aktuelle Dialogsituation ins Bewußtsein des Benutzers zu heben. Der Bezug auf die im statischen und dynamischen Dialogzustand gegebene Information ist dabei nicht so schwierig herzustellen wie das unmittelbare und mühelose Verständnis des Benutzers. Durch Rückfragen und Fehlermeldungen kündigen sich Fehlersituationen an. Aber die in Meldungen enthaltenen Namen und Bezeichnungen von Kernobjekten stellen manchmal nur einen Folgefehler dar. Es ist eine auch bei der Programmierung im Stapelbetrieb gemachte grundsätzliche Erfahrung, daß sich eigentliche Fehlerursachen im normalen Ablauf verbergen und erst durch Folgefehler zeigen. Die Verständigung ist dann schwierig. Um den Benutzer nicht in Ärger oder Wut zu versetzen, muß ein Dialogsystem seine Fehlermeldungen zurückhaltend formulieren.

Aber auch die Darstellung direkt erkannter Fehler ist schwierig genug. Zwischen einem Implementierer, der bei seiner Arbeit auf kritische Systemzustände stößt und sich die Darstellung eventueller Fehler überlegt, und einem Anwender, der das Dialogsystem später benützt und nicht kennt, tut sich eine tiefe Kluft auf: Sie betrachten und beschreiben einen Fehler und seine Ursache von verschiedenen Standpunkten aus. Man kann die beiden mit zwei Menschen vergleichen, die sich gegenüberstehen und etwa einen Mast zwischen sich haben. Sie sollten mit dem Mast in einer Linie stehen und haben dann gleichsam den Dialogzustand beide richtig erfaßt. Die beiden bewegen sich abwechselnd, und plötzlich befindet sich der Mast nicht mehr auf ihrer Verbindungslinie. Wie beschreiben beide Partner nun die Situation? Der eine sagt: "Du stehst zu weit links",

und der andere: "Du stehst zu weit rechts". Trotzdem können sich die Partner im Vergleich helfen. Sie sehen sich, durchschauen die Lage und verständigen sich einfacher und eindeutig mit Armbewegungen. Im Dialog bleibt aber nur die verbale Verständigung, und wenn der eine Partner rät: "Geh weiter nach rechts!", und der andere Partner geht dahin, wo für ihn rechts ist, so reagiert er gerade falsch. Wenn der Implementierer die Texte für Fehlermeldungen und Korrekturvorschläge entwirft, hat er es nun nicht so einfach, daß er nur Seitenangaben vertauschen muß. Oft ist der Irrtum des Benutzers nur zu vermuten. Dann kann diese Vermutung ausgegeben werden. Manchmal liefert der Probetrieb statistische Wahrscheinlichkeiten für verschiedene Fehlerursachen, so daß die Texte der Meldungen danach gestaltet werden können. Bei seltener auftretenden Fehlersituationen müssen Korrekturhinweise die momentane Ratlosigkeit des Benutzers überbrücken.

4.3.2.3 Die Sprache bei EBIDIS

Sprachliche Systemäußerungen werden im momentan implementierten Umfang des Dialogsystems aus über 70 konstanten Textteilen gebildet. Einige davon werden untereinander kombiniert oder mit Benutzernamen zusammengesetzt. Bei den Benutzereingaben unterscheidet das Dialogsystem

- Kommandos,
- Anweisungen in Bildsprache und
- Anweisungen in der Rahmensprache.

Eine Eingabe wird zunächst der Kommandoentschlüsselung unterworfen. Ein erkanntes Kommando wird sofort ausgeführt. Sonst wird die durch lexikalische Analyse vorverarbeitete Eingabe erst dem Bildsprachenübersetzer angeboten und - falls dieser Symbole der Rahmensprache erkennt - dem Rahmenübersetzer. Die Unterscheidung der Eingabesprache in drei Anteile wird nur vom Dialogsystem vorgenommen. Für den Benutzer erscheint sie

als ein einheitliches Ganzes. Die Sprachanteile sind auch nicht disjunkt. Z. B. ist die Eingabe

ZEIGE BILD ANSICHT

eine Bildsprachenanweisung oder ein Kommando je nachdem, ob diese Textzeile hinterlegt wird oder nicht. Auch der Aufbau der Rahmensprachenanweisungen ist dem von Kommandos sehr ähnlich.

4.3.2.3.1 Kommandos

Kommandos bilden den Teil der Eingabesprache, welcher dem Dialogsystem am stärksten zugewandt ist. Die Kernsprache zur Bildbeschreibung und Bildbearbeitung ist mehr am Anwenderproblem orientiert. Bei [KuWi 75] wird stattdessen von sekundären und primären Sprachelementen gesprochen. Die EBIDIS-Rahmensprache nimmt eine Mittelstellung ein.

Wegen der Systemnähe von Kommandos mußten auch bei der Beschreibung von Verwaltungstätigkeiten (Abschnitt 4.3.1) ständig Kommandos als Beispiele benutzt werden. Sie aktivieren die Algorithmen des Dialogsystems unmittelbar. Die in Abschnitt 4.3.1 genannten Datenbereiche mit ihren individuellen Strukturen erfordern unterschiedliche Bearbeitung. Ein Teil des bis jetzt implementierten Dialograhmens ist nach Datenbereichen gegliedert und weiter nach Tätigkeiten auf diesen Bereichen. Einfache Dialoge lassen die hierarchische Gliederung des Dialogsystems noch erkennen. Dafür folgen zwei Beispiele, in denen die Benutzereingaben wieder unterstrichen sind:

GIB KOMMANDO, ANWEISUNG ODER "?"

: NAME

NAMEN "NEU" DEFINIEREN, "ZEIGEN" ODER "LOESCHEN" ?

: NEU

TYP "INT" ODER "PIC" UND LISTE NEUER NAMEN

> PIC B(2Ø)

VERSTANDEN

TYP "INT" ODER "PIC" UND LISTE NEUER NAMEN

>

NAMEN "NEU" DEFINIEREN, "ZEIGEN" ODER "LOESCHEN" ?

:

GIB KOMMANDO, ANWEISUNG ODER "?"

: TEXT

TEXT N(EUSCHREIBEN), Z(EIGEN), S(PEICHERN), H(OLEN)

TEXT K(OMPRIMIEREN) ? [ODER L(OESCHEN) ?

: ZEIGEN

TEIL > ALLE

KEIN TEXT VORHANDEN

TEXT N(EUSCHREIBEN), Z(EIGEN), S(PEICHERN), H(OLEN)

TEXT K(OMPRIMIEREN) ? [ODER L(OESCHEN) ?

:

Diese Dialogabschnitte zeigen die einfache Bedienung eines hierarchisch gegliederten Systems und ähneln dabei mehr einem Ballspiel als einer Unterhaltung. Darum gestattet EBIDIS dem Benutzer, mehrere Eingaben zusammenzufassen. Obige Beispiele kürzen sich dann ab:

GIB KOMMANDO, ANWEISUNG ODER "?"

: NEUE PICTURE NAMEN P, Q, R, S, T

VERSTANDEN

TYP "INT" ODER "PIC" UND LISTE NEUER NAMEN

>

NAMEN "NEU" DEFINIEREN, "ZEIGEN" ODER "LOESCHEN" ?

:

GIB KOMMANDO, ANWEISUNG ODER "?"

: ALLEN TEXT ZEIGEN

KEIN TEXT VORHANDEN

TEXT N(EUSCHREIBEN), Z(EIGEN), S(PEICHERN), H(OLEN)

TEXT K(OMPRIMIEREN) ? [ODER L(OESCHEN) ?

:

So wird zwar die hierarchische Struktur des Dialogsystems verwischt, aber die zusätzliche Möglichkeit solcher Eingaben kommt den Wünschen eines geübten Benutzers entgegen. Er registriert eine gewisse Intelligenz beim Gegenüber, fühlt sich nicht als dumm betrachtet und ermüdet nicht so leicht. Nun erfordert die vollständige Analyse von Eingaben in natürlichem Deutsch einen beträchtlichen Aufwand (vgl. [GuLö 77]). Darum wurden bei EBIDIS einfache Mittel eingesetzt, um bei den Eingaben doch eine beträchtliche Nähe zur natürlichen Ausdrucksweise des Benutzers zu erreichen. Die reservierten Wortsymbole haben jeweils mehrere externe Darstellungen mit verschiedenen grammatikalischen Formen. Beispiele dafür sind ALLE und ALLEN oder NAME und NAMEN oder NEU, NEUE und NEUER. Die Menge der gültigen Darstellungen eines Wortsymbols kann dabei leicht geändert werden. Das Abbilden verschiedener Formen z.B. eines Wortsymbols auf eine interne Darstellung kann man als Freiwortanalyse bezeichnen. Weiterhin darf der Benutzer die Reihenfolge der Worte im Kommando frei wählen. Bei der Kommandoentschlüsselung handelt es sich also nicht mehr um eine Syntaxanalyse im gewohnten Sinn, sondern um eine inhaltliche Analyse. Es wird untersucht, ob die Eingabe ein Wortsymbol enthält, das einen Datenbereich kennzeichnet. Das wäre dann bereits ein gültiges Kommando. Weiterhin wird die ganze Eingabe nach Wortsymbolen durchsucht, die vielleicht auch schon eine Tätigkeit und einen Datentyp bezeichnen, und nach und nach Benutzernamen. Werden solche Angaben gefunden, so unterbleiben diesbezügliche Fragen und Eingabeanforderungen des Dialogsystems. Welche Ausdrucksmöglichkeiten und Varianten von Kommandos einem Benutzer dann zur Verfügung stehen, zeigen folgende Beispiele:

ZEIGE ALLE NAMEN
ALLEN TEXT ZEIGEN
NEUER NAME
NAMEN A, C LOESCHEN
NEUE INTEGER NAMEN A, B, C(50)
KOMPRIMIERE TEXT

Es ist für den Benutzer kaum zu behalten, ob nun etwa ZEIGE ALLE NAMEN oder ALLE NAMEN ZEIGEN die richtige Form des Kommandos war. Die Erleichterung bei der Formulierung von Eingaben ist damit offensichtlich. Das erklärt auch, warum die in Abschnitt 4.3.1 verwendeten Kommandos meistens mit Zusätzen wie "etwa" oder "z.B." angegeben wurden. Feste Formulierungen werden nicht verlangt. Auch der Umfang eines Kommandos ist frei wählbar und paßt sich der Übung und Konzentration des Benutzers an. Der Nachteil dieser Art von Kommandoentschlüsselung liegt darin, daß auch falsche Endungen eingegeben werden können, die nicht zurückgewiesen werden. Ein großer Vorteil ist auch die Flexibilität und Anpassungsfähigkeit der Kommandoanalyse. Bei der Eingabe von

ZEIGE ALLE RESERVIERTEN NAMEN

werden z.B. nur die reservierten und nicht etwa alle Namen aufgelistet. Eine Kurzbeschreibung von Kommandos kann der Benutzer auch im Dialog erhalten:

GIB KOMMANDO, ANWEISUNG ODER "?"

: ?

KOMMANDO: ENTHAELT WENIGSTENS "TEXT" ODER "NAME",
DARF AUCH ENTHALTEN: ["HOLE", "KOMPR(IMIERE)",
EINE AKTION "ZEIGE", "NEU", "LOESCHE", "SPEICHERE",
EINEN TYP "INT(EGER)", "PIC(TURE)" ODER "ALLE"
UND EINZELNE NAMEN

MIT FREIEN WORTEN IN BELIEBIGER REIHENFOLGE

ANWEISUNG: IN BILD- ODER RAHMENSPRACHE

GIB KOMMANDO, ANWEISUNG ODER "?"

4.3.2.3.2 Bildsprache

Die primären Sprachelemente einer Dialogsprache umfassen die problemorientierten Datenstrukturen - sie sind bei EBIDIS in der Namensliste und in der Bildbank zusammengefaßt - und die zugehörigen Operationen darauf. Die formale Grundlegung einer bildbeschreibenden Sprache, welche auch die Strukturen der betrachteten Bilder klar herausstellt, geschah in einer Arbeit von Shaw ([Sh 69]). Die dort definierte *Picture Description Language* PDL enthält Ausdrücke, die arithmetischen Ausdrücken sehr ähnlich sind. Eine Realisierung dieses formalen Ansatzes ist die Bildprogrammiersprache EHBIS ([Br 76]). Die bei der Implementierung des Präcompilers und des umfangreichen Laufzeit-Systems für EHBIS gemachten Erfahrungen sowie die möglichen Verbesserungen, die sich aus dem praktischen Einsatz von EHBIS ergaben, werden bei der Verwirklichung des Dialogkernes zugrundegelegt.

4.3.2.3.2.1 Primitive Bilder

Die Bildsprache bei EBIDIS stellt konstante Bildelemente, sogenannte "primitive Bilder" zur Verfügung (Figur 11). Hinzu kommen noch Dunkellinien und eventuell die bei EHBIS implementierten Funktionen und Parameterfunktionen. Die



Fig. 11: Primitive Bilder in EBIDIS

Bezeichnungen für primitive Bilder stehen jeweils für eine ganze Klasse von Bildelementen, deren einzelne Repräsentanten sich nur durch Merkmale wie Größe, Helligkeit, Drehlage oder Strichart voneinander unterscheiden. Ist in einer EBIDIS-Anweisung ein primitives Bild enthalten, etwa

... LINIE [GROESSE = 100, RICHTUNG = 45] ...

so wird zur Ausführungszeit ein strukturiertes Datum von der Art *primbild* in der Bildbank erzeugt. Es stellt dieses primitive Bild mit den Merkmalswerten dar, die entweder explizit vorgeschrieben wurden oder standardmäßig gewählt werden. Für jede Klasse primitiver Bilder ist ein Bildcode-generator vorhanden, der beim ersten Zeigen des primitiven Bildes das zugehörige Programmstück im Bildwiederholtspeicher erzeugt.

Bei allen primitiven Bildern sind zwei Punkte ausgezeichnet für die geometrische Verknüpfung untereinander. Die Lage dieser Punkte ist für die einzelnen Klassen primitiver Bilder in Figur 11 durch einen gestrichelten Pfeil angegeben. Die Größe und Richtung eines Bildes werden an diesem Pfeil abgelesen. Er ist normalerweise nicht sichtbar, wenn ein Bild am Bildschirm gezeigt wird, kann aber durch Tastendruck sichtbar gemacht werden (s. Abschnitt 4.3.2.3.4).

4.3.2.3.2.2 Verknüpfende Operatoren

Der im eindimensionalen Fall gegebenen Konkatenation von Zeichenketten stehen bei der Beschreibung von zweidimensionalen Bildern jedoch wesentlich mehr Möglichkeiten der gegenseitigen Anlagerung von Teilbildern gegenüber. Figur 12 zeigt die Verknüpfung von abstrakten Teilbildern durch vier verschiedene binäre Operatoren (s.u.). Durch Bildausdrücke entstehen zusammengesetzte Bilder, in Figur 12 als gewellte Pfeile dargestellt. Auch sie haben zwei ausgezeichnete Punkte, die ihnen von den beiden Bildoperanden

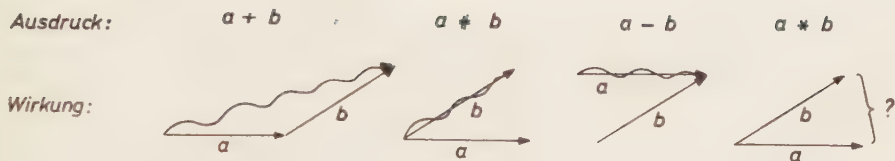


Fig. 12: Binäre Bildoperation bei EBIDIS

vererbt wurden. Dadurch kommen auch zusammengesetzte Bilder selbst wieder als Bildoperanden in Frage. Zusammengesetzte Bilder werden in der Bilddbank als binäre Bäume dargestellt. Sie werden durch ein strukturiertes Datum von der Art *operator* repräsentiert, das dem zuletzt ausgeführten Operator entspricht. Die Art *operator* enthält u.a. die Verweise auf die beiden Bildoperanden. Eine ähnliche Struktur für Bilddaten findet sich in [En 76], dort als "strukturiertes Format" bezeichnet.

Besondere Erwähnung verdient noch die Bildverknüpfung mit $*$. Hier werden die abstrakten Bilder kongruent verschmolzen. Ist das nicht von vorneherein möglich, so wird der rechte Bildoperand angepaßt: Richtung und Größe des linken Bildoperanden werden auch dem rechten aufgezwungen, ein Seiteneffekt, der einerseits dem Dialogsystem sehr viel Anpassungsarbeit verursacht, besonders bei umfangreichen Bildoperanden, der aber andererseits dem Benutzer eine einfache Beschreibung von Bildkonstruktionen erlaubt.

Es gibt in EBIDIS noch zwei unäre Operatoren:

- $\leftarrow a$ vertauscht die beiden ausgezeichneten Punkte des Bildes a , kehrt also den Pfeil des abstrakten Bildes um,
- $/a$ erzeugt eine Dunkellinie mit der Länge und Richtung des Bildes a .

Treten in einem Bildausdruck mehrere Operatoren auf, so werden

zuerst $\leftarrow$ und $/$,
dann $*$,
dann $+$, $-$ und $*$

abgearbeitet. Operatoren gleicher Priorität werden von links nach rechts abgearbeitet, abweichende Reihenfolgen gibt der Benutzer durch runde Klammern an.

Die häufigste Form einer Bildanweisung ist dann die Zuweisung eines Bildes, gegeben als primitives Bild oder als Bildausdruck, an eine Bildvariable, z.B.

`FIC1 = LINIE * BOGEN(90) + PART2`

Der Kern von EFIGIS enthält auch einfache Größen vom Typ *Integer*, bei deren Auftreten durch obige Operatoren die üblichen Operationen der Ganzzahlarithmetik beschrieben sind.

4.3.2.3.2.3 Bildmanipulationen

Die in Abschnitt 4.3.2.3.2.1 gezeigte Merkmalsliste kann auch bei einem Bildnamen oder Bild-Zwischenergebnis angegeben werden. Ein Beispiel dafür ist

`ALPHA = (BETA[RICHT=135] + GAMMA) [GR = 300]`

mit abgekürzten Merkmalsberechnungen. Es bestehen nun Wechselbeziehungen zwischen den Bildmerkmalen 'Größe' und 'Richtung' und den Bildoperatoren. Bildoperatoren verändern oder konstituieren Merkmale, und die Zuordnung von Merkmalen zu Bildern kann nicht ohne Rücksicht auf die Bildstruktur vorgenommen werden, welche sich aus den verwendeten Bildoperatoren ergibt. Diese Wechselbeziehungen wurden bei EFIGIS so eingerichtet, daß eine bequeme und konsistente Handhabung von Bildern durch Verknüpfungen

und explizite Merkmale gewährleistet ist ([Br 76]).

Eine Schwierigkeit ergab sich beim Entwurf der EHBIS-Semantik von Bildmanipulationen und Bildzuweisungen. Diese Schwierigkeit ist im Bereich der VHL-Programmiersprachen bekannt ([Schw 75]) als "copy problem", eine Kontroverse zwischen "value semantics" und "pointer semantics". Sie tritt dort auf, wo an umfangreichen Datenstrukturen Änderungen vorgenommen werden und wo solche Objekte Variablen zugewiesen werden. Man möchte das umfangreiche Kopieren von Daten vermeiden, muß aber unerwünschte Seiteneffekte verhindern. Bei EHBIS wurde das Problem durch zusätzliche Vermerke in der Bildbank gelöst: Es ist bei jedem Bild feststellbar, ob es Teilbild eines anderen Bildes ist oder nicht. Teilbilder werden vor einer weiteren Verwendung in der Bildbank kopiert, andere Bilder nicht ([Br 76]).

Zusätzlich zu den bei EHBIS gegebenen Möglichkeiten sind weitere Anordnungsoperatoren denkbar, die eine Anzahl von Bildteilen ohne sichtbaren Zusammenhang zyklisch oder rechtwinklig zu Gruppen anordnen und die quadratische Darstellungsfläche des Bildschirms aufteilen. Insbesondere soll die zusätzliche rechteckige Darstellungsfläche des Bildschirms auch für Objektbilder wie z.B. zu entwerfende Bildschirm-Menüs verfügbar sein. Auch das allgemeine Wiedergewinnen von Bildmerkmalen aus der Bildbank muß die Bildsprache von EBIDIS ermöglichen.

4.3.2.3.3 Rahmensprache

Die Anweisungen der Bild- und Rahmensprache werden im Gegensatz zu den Kommandos syntaktisch untersucht. Bildanweisungen haben meistens die Form einer Wertzuweisung mit einem Bildausdruck als rechter Seite, Rahmenanweisungen erlauben eine Abarbeitungsreihenfolge von Anweisungen, die von der in Abschnitt 4.3.1.3 beschriebenen implizit gegebenen Reihenfolge

abweicht. Bei Programmiersprachen leisten das die Kontrollanweisungen wie Laufschleifen oder Fall-Anweisungen mit ihren vielfältigen Varianten. Bei EBIDIS werden solche Anweisungen in die gesamte Dialogsprache integriert. Sie erscheinen darum auch nicht mit den geläufigen Schlüsselwörtern *for*, *do*, *if*, *then* oder *else*, sondern mit deutschen Wortsymbolen und in der Form der Kommandos. Die wichtigste Anweisung dient dem Starten von Aufträgen oder dem geschachtelten Aufruf von Anweisungen, z.B.

RECHNE TEIL 2.1.4

Diese Grundanweisung kann mit Zusätzen versehen sein. Z.B. werden die Anweisungen

MIT A = 3, C = 7 RECHNE TEIL 5.2.7

oder

RECHNE ... MIT ...

nach vorheriger Wertzuweisung ausgeführt. Die dabei gesetzten Größen dienen als Parameter zur Prozedur des Teiles 5.2.7, die jedoch global übergeben werden. Eine Laufschleife hat bei EBIDIS die Form

FUER LV = A(S)E RECHNE ...

wobei A, S und E für Anfangswert, Schrittweite und Endwert stehen. Wenn EBIDIS später auch Boole'sche Größen und Vergleiche umfaßt, können bedingte Anweisungen wie z.B.

WENN G < 500 GILT, DANN RECHNE TEIL 3.4

WENN B GILT, DANN RECHNE TEIL 2, SONST RECHNE TEIL 5

oder

RECHNE ..., WENN ...

eingegeben werden. Wichtig ist auch die von BLISS ([Wu 70]) her bekannte Abbruchanweisung. Bei EBIDIS werden keine Kontrollbereiche unterschieden (EXIT COMPOUND, EXIT CONDITIONAL u. a.), sondern die zu beendende Einheit wird immer angegeben, z.B.

BEENDE TEIL 3.1.2

Wird solch eine Anweisung erst hinterlegt und dann ausgeführt, so veranlaßt sie die Ausführung der logisch folgenden oder der umgebenden Anweisung. Wird sie direkt eingegeben, so löscht sie einen Teil der im Auftragskeller aktivierten Anweisungen einschließlich der bezeichneten Anweisung, und die weitere Abarbeitung der gekellerten Aufträge muß durch das Kommando WEITER angestoßen werden. Mit der Abbruchanweisung ist es insbesondere auch möglich, eine Endlos-Anweisung der Form

RECHNE TEIL ... DAUERND

oder eine Laufschleife ohne obere Grenze zu beenden.

4.3.2.3.4 Nichtsprachliche Verständigung

Die nichtsprachliche Verständigung bei einem EBIDIS-Dialog dient der Erleichterung der Dialogführung und einer gewissen Abrundung. Für die rasche Bildausgabe wird ein graphisches Sichtgerät VT 15 von DEC benützt. Es verfügt noch über einen Lichtgriffel, zwei Steuerknüppel und programmierbare Funktionstasten. Ruft der Benutzer nach dem Kommando

NEUES BILD

ein schon vorhandenes Bild als Operanden auf, so kann über die Funktionstasten das helle Zeichnen aller schwach oder gar nicht sichtbaren Linien veranlaßt werden und gezielte Bildänderungen durch Lichtgriffeleinsatz eingeleitet werden. Die schrittweise selektierende Teilbildidentifikation durch Lichtgriffel bei EHBIS sei hier erwähnt, ebenso das Bildschirm-Menü für die möglichen Bildänderungen. Das Bildwechseln durch Funktionstastendruck bei der Inspektion von mehreren Bildern nacheinander wurde schon in Abschnitt 4.3.1.2 beschrieben. Werden nach dem Kommando NEUES BILD zwei Bildoperanden aufgerufen und gleichzeitig abgebildet, so dienen die beiden Steuerknüppel dem probeweisen Verschieben beider Operandenbilder gegeneinander.

Die Technik des *menue picking* ist grundsätzlich für die Abwicklung des ganzen sprachlichen Dialoges geeignet. Dann ist aber weder das Zusammenfassen mehrerer Eingaben möglich - eine wichtige Eigenschaft bei EBIDIS - noch das direkte Protokollieren des Dialoges.

Die nichtsprachlichen Systemäußerungen bestehen einmal in der weitergehenden Anzeige des dynamischen Dialogzustandes und zum anderen in akustischen Signalen. Die sechs programmierbaren Funktionstasten haben je zwei Zustände, die durch Erleuchten der Tasten bzw. durch Erlöschen auch angezeigt werden. Dann ist es möglich, die einzelnen Systemtätigkeiten innerhalb eines Dialogschrittes in den Funktionstasten - gleichsam eine Miniaturkonsole beim Sichtgerät - laufend anzuzeigen. Diese Anzeige ist für den geübten Benutzer eine zusätzliche Dialogüberwachung. Denkbar sind folgende Einzelanzeigen:

- der Bildsprachenübersetzer ist aktiv
- der Rahmensprachenübersetzer ist aktiv
- Auftragsabwicklung allgemein
- insbesondere Bearbeitung der Bildbank
- Erzeugung von Bildcode
- das Dialogsystem ist untätig

Während der Eingabe abzulegender Textzeilen sind die beiden Übersetzer im Hintergrund tätig. Die Auftragsabwicklung findet im Vordergrund statt. Hierbei sind die Übersetzer normalerweise nicht mehr aktiv. Sie werden nur aufgerufen, wenn eine Anweisung direkt eingegeben wurde oder wenn wegen Platzmangels der Code für zu durchlaufende Anweisungen überschrieben wurde. Vor Ausgabe eines Bildes auf Bildschirm wird aber der ganze Code für das Bild erzeugt. Die Codeerzeugung während der Bildausgabe würde ein unzumutbares Flackern des Bildes bewirken.

Die akustischen Signale dienen einer besonders einfachen Dialogüberwachung. Entsprechen die Systemäußerungen im normalen Dialogverlauf den Erwartungen des Benutzers, so wird die nächste Eingabeanforderung von einem leisen Anschlag des Blindzeichens auf dem Fernschreiber begleitet. Bei Abweichungen vom normalen Dialogverlauf ertönt jedoch ein Klingelzeichen.

5 BEMERKUNGEN ZUR IMPLEMENTIERUNG

5.1 Das EHBIS-Programmiersystem

Die Entwicklung der Erlanger Bildsprache begann im Jahre 1974. Bei der ersten Implementierung auf dem Graphik-Rechner des Institutes wurde FORTRAN als Zwischensprache gewählt. Ein Präcompiler übersetzt EHBIS-Programme in FORTRAN-Anweisungen, welche die Routinen des EHBIS-Laufzeitsystems aufrufen. Diese EHBIS-Programme können auch bereits FORTRAN-Zeilen enthalten für numerische Berechnungen, Laufschleifen und sonstige Angaben zum Programmfluß. Eingestreute FORTRAN-Zeilen werden vom EHBIS-Präcompiler in das Zwischenprogramm unverändert übertragen. Entsprechend der Ausstattung des Graphik-Rechners ist die erste Version eine interaktive Implementierung, die den Lichtgriffel des Sichtgerätes zur Laufzeit sowohl des EHBIS-Systems wie auch der erzeugten Objektbilder einsetzt.

Diese erste Implementierung wurde im Rahmen zweier Diplomarbeiten durchgeführt ([Ru 75],[Leh 75]). Da der CDL-15-Übersetzer damals noch nicht zur Verfügung stand, wurde FORTRAN als Implementierungssprache gewählt. Die Quelle des EHBIS-Präcompilers umfaßt etwa 2800 Zeilen, die des EHBIS-Laufzeitpaketes etwa 1500 Zeilen FORTRAN-Text und ebensoviele Zeilen Assemblercode.

Eine Erweiterung des EHBIS-Programmiersystems ist die Plotter-Ausgabe der am Bildschirm interaktiv erstellten und geprüften Bilder. Dazu wurde die erste EHBIS-Implementierung auf einen Großrechner (TR 440) übertragen mit einem Benson-Plotter in der Peripherie. Wegen des Fehlens interaktiver Arbeitsmöglichkeiten an diesem Ausgabegerät wurde das EHBIS-Laufzeitsystem um die Routinen für Lichtgriffeleingabe und -auswertung gekürzt und mit Anpassungen versehen, wo die Wiedergabemöglichkeiten des Plotters hinter denen des graphischen Sichtgerätes zurückstehen

(Strichelung). Überlegenheiten des Plotters wie schräg geschriebener Text und größere Zeichenfläche wurden dagegen ausgenützt. Diese zweite EHBIS-Implementierung wurde ebenfalls als Diplomarbeit durchgeführt ([Wo 76]). Sie enthält nicht nur die Ausgabefunktionen, sondern auch alle Möglichkeiten der Bildkonstruktion und -manipulation in EHBIS. Der Präcompiler wurde dazu neu geschrieben in CDL und umfaßt etwa 750 Zeilen. Er übersetzt EHBIS-Programme wieder in die Zwischensprache FORTRAN.

5.2 Das Dialogsystem EBIDIS

Das in Kapitel 4 beschriebene Dialogsystem wurde bereits teilweise auf dem Graphik-Rechner des Instituts, einer Anlage PDP-15 mit einem graphischen Sichtgerät VT-15, in CDL implementiert. Die Änderungsfreundlichkeit von CDL-Programmen erlaubt, daß auch teilweise fertiggestellte Systeme während der fortschreitenden Realisierung ständig lauffähig sind und getestet werden können.

Dem hierarchischen top-down-Entwurf des globalen Systemkonzepts entsprechend sind Programmteile an der Spitze des Programmbaumes bereits realisiert. Dazu gehören die in Abschnitt 4.3.1.5 beschriebene Dialogverwaltung, die Vorbereitung und die Abschlußbehandlung eines Dialoges. Von den Tätigkeiten innerhalb eines Dialoges sind alle Kommandos und Operationen auf dem Textspeicher verwirklicht. Die Hintergrundarbeit des Dialogsystems besteht bis jetzt im ständigen Sichern des Textspeichers nach Änderungen. Die Namensliste ist mit ihrer internen Verwaltung realisiert und mit allen Kommandos zu den Tätigkeiten *neu*, *zeige* und *lösche*. Die Programmquellen des Dialogsystems umfassen im jetzigen Implementierungsstadium etwa 1500 CDL-Zeilen und 700 Assemblerzeilen.

6 DANK

Herrn Prof. Dr. Schneider danke ich für die Betreuung dieser Arbeit und meinen Kollegen M. Nagl, W. Altmann, H. Göttler, G. Stöth und D. Weber vom Lehrstuhl für Programmiersprachen für ihre zahlreichen, wertvollen Anregungen und Hinweise.

Besonders danke ich auch meiner Frau Roswitha, meiner Schwester Frau G. Müller-Osten und Frau E. Lührs für das sorgfältige Schreiben dieser Arbeit und Herrn A. Wanninger für die gewissenhafte Anfertigung der Zeichnungen.

7 LITERATURVERZEICHNIS

- [AlWe 77] ALTMANN, W./WEBER, D.: "Programmiermethodik", Arbeitsbericht des IMMD, Bd. 10, Nr. 3, Erlangen, 1977.
- [Br 77] BRÄUER, I.: "Ein Kreuzcompiler der Sprache PASCAL für die Wirtsmaschine TR 440 mit Laufzeitsystem für die Zielmaschine INTERDATA 7/32", Diplomarbeit am Lehrstuhl für Programmiersprachen, Erlangen, 1977.
- [Br 74] BRÜGMANN, H.-W.: "Das Programmieren von Compilern in CDL", interner Bericht des Lehrstuhls für Programmiersprachen, Erlangen, 1974.
- [Br 75] BRÜGMANN, H.-W.: "Beschreibung eines SLR(1) Parsing Table Constructor", interner Bericht des Lehrstuhls für Programmiersprachen, Erlangen, 1975.
- [Br 76] BRÜGMANN, H.-W.: "EBBIS: High Level Picture Programming", in: Schneider, H.-J./Nagl, M. (Eds.): "Programmiersprachen", Informatik-Fachbericht Bd. 1, Springer, 1976.
- [CoGr 73] CONWAY, R./GRIES, D.: "An Introduction to Programming", Winthrop Publishers, 1973.
- [De 75] DEHOTAY, J.-P./KOSTER, C. H. A./STAHL, H.-M.: "Syntaktische Beschreibung von CDL 2", Bericht der Forschungsgruppe Programmiersprachen und Compiler II, TU Berlin, 1975.
- [DDH 72] DAHL, O.J./DIJKSTRA, E.W./HOARE, C.A.R.: "Structured Programming", Academic Press, 1972.
- [Dzi 77] DZIDA, W./HERDA, S./ITZFELDT, W.D./SCHUBERT, H.: "Zur Benutzerfreundlichkeit von Dialogsystemen", Bericht Nr. 35 des Instituts für Software-Technologie, GMD St. Augustin, 1977.
- [En 76] ENCARNACAO, J./ECKERT, R.: "Bemühungen und Möglichkeiten bei der Begriffsbildung und Normung graphischer Systeme", in: ACM German Chapter, Lectures II, 1976.
- [En 77] ENCARNACAO, J./FINK, B./HÖRBST, E./KONKART, R./NEES, G./PARNAS, D.L./SCHLECHTENDAHL, E.G.: "A Recommendation on Methodology in Computer Graphics", Bericht KFK 2394 der Gesellschaft für Kernforschung m.b.H., Karlsruhe, 1977.

- [Gr 74] GRIFFITHS, M.: "LL(1) Grammars and Analysers", Lecture Notes 2B of the Advanced Course on Compiler Construction, TU München, 1974.
- [GuLö 77] GUNTERMANN, R./LÖTSCHER, A.: "Problemlösendes Informationssystem mit Deutsch als Interaktions-sprache", Artikel in: Computerwoche Nr. 35, 1977.
- [HeJä 77] HEYDERHOFF, P./JÄHNICHEN, S.: "Mit ELAN kinderleicht programmieren lernen", Artikel in: Computerwoche Nr. 20, 1977.
- [HeVo 75] HENNING, W./VOLKERT, J.: "Ein interaktives kartographisches Auskunftssystem", in: Nachrichten aus dem Karten- und Vermessungswesen, Reihe I: Originalbeiträge, Heft Nr. 68, Verlag des Instituts für Angewandte Geodäsie, Frankfurt/M., 1975.
- [Ho 69] HOARE, C.A.R.: "An Axiomatic Basis for Computer Programming", in: Comm. ACM 12, Nr. 10, 1969.
- [Ho 75] HOLAGER, P.: "Some Program Structuring Rules applied to an Example" in: Cousot, P. (Ed.): MOL Bulletin Nr. 6, IFIP Working Group 2.4, 1977.
- [Hor 74a] HORNING, J.J.: "LR Grammars and Analysers", Lecture Notes 2C of the Advanced Course on Compiler Construction, TU München, 1974.
- [Hor 74b] HORNING, J.J.: "What the Compiler should tell the User", Lecture Notes 5D of the Advanced Course on Compiler Construction, TU München, 1974.
- [JeWi 75] JENSEN, K./WIRTH, N.: "PASCAL User Manual and Report", Springer, 1975.
- [Kn 71] KNUTH, D.E.: "Top-down Syntax Analysis" in: Acta Informatica 1, 1971.
- [Ko 71a] KOSTER, C.H.A.: "Affix Grammars", in Peck, J.E.L. (Ed.): "Algol 68 Implementation", North-Holland Publishing Company, Amsterdam, 1971.
- [Ko 71b] KOSTER, C.H.A.: "A Compiler Compiler", Report MR 127, Mathematisch Centrum Amsterdam, 1971.
- [Kog 76] KOGON, R./LATTERMANN, D./LEHMANN, H./OTT, N./ZOEPPRITZ, M.: "The User Specialty Languages System", in: Neuhold, E.J. (Ed.): Tagungsbericht der 6. GI-Jahrestagung, Informatik-Fachbericht Bd. 5, Springer, 1976.

- [KräLo 74] KRÄGELOH, K.-D./LOCKEMANN, P.C.: "Schichten von Datenmodellen und Datenbanksprachen", in: Falkenberg, E./Klutentreter, W. (Eds.): Sommerseminar "Datenmodelle und Systementwürfe für Datenbanksysteme", GMD St. Augustin, 1974.
- [KuWi 75] KUPKA, I./WILSING, N.: "Dialogsprachen", Teubner, 1975.
- [Le 74a] LEAVENWORTH, B.M. (Ed.): "Proceedings of a Symposium on Very High Level Languages", ACM SIGPLAN Notices, Bd. 9, Nr. 4, 1974.
- [Le 74b] LEAVENWORTH, B.M.: "Nonprocedural Programming", in: Hackl, C.E. (Ed.): "Programming Methodology", LNCS 23, Springer, 1975.
- [Leh 75] LEHR, H.: "Ein Programmpaket zur interaktiven Definition und Manipulation von strukturierten, dialogfähigen Bildern zweidimensionaler Objekte", Diplomarbeit am Lehrstuhl für Programmiersprachen, Erlangen, 1975.
- [Lu 61] LUCAS, P.: "Die Strukturanalyse von Formelübersetzern", in: Elektronische Rechenanlagen 3, 1961.
- [Mau 69] MAURER, H.: "Theoretische Grundlagen der Programmiersprachen", BI-Hochschultaschenbuch Bd. 404/404a*, Bibliographisches Institut, Mannheim, 1969.
- [Mu 74] MUTH, D.: "Erstellung strukturierter Programme im Dialog", in: ACM German Chapter, Lectures III, 1974.
- [Po 74] POOLE, P.C.: "Portable and Adaptable Compilers", Lecture Notes 5A of the Advanced Course on Compiler Construction, TU München, 1974.
- [Roe 76] ROEHRICH, J.: "Syntax-Error Recovery in LR-Parsers", in: Schneider, H.-J./Nagl, M. (Eds.): "Programmiersprachen", Informatik-Fachbericht, Bd. 1, Springer, 1976.
- [Ru 75] RUDERT, A.: "Ein Präcompiler für EHBIS, eine höhere Sprache zur Programmierung von Bildern und Bildschirmdialogen", Diplomarbeit am Lehrstuhl für Programmiersprachen, Erlangen, 1975.
- [Schw 75] SCHWARTZ, J.T.: "The Copy Problem in SETL and the Destructive Use Condition", in: Proceedings of an International Summer School on Program Analysis and Optimization, IBM Israel Scientific Center, Haifa, 1975.

- [Sh 69] SHAW, A.C.: "A Formal Picture Description Scheme as a Basis for Picture Processing Systems", in: Information and Control 14, 1969.
- [Sn 75a] SCHNEIDER, H.-J.: "Compiler", de Gruyter, 1975.
- [Sn 75b] SCHNEIDER, H.-J.: "Theoretische Untersuchungen zur strukturierten Programmierung", Vorlesungsmanuscript, Erlangen, 1975.
- [vWijn 69] van WIJNGAARDEN, A. (Ed.): "Report on the Algorithmic Language Algol 68", in: Numerische Mathematik 14, 1969.
- [Wi 71] WIRTH, N.: "Program Development by Stepwise Refinement", in: Comm. ACM 14, 1971.
- [Wo 76] WOLF, K.: "CDL-Implementierung der Erlanger Höheren Bildsprache für die Konstruktion und Einschleusung strukturierter Bilder auf der Rechenanlage TR 440 und Ausgabe auf Plotter", Diplomarbeit am Lehrstuhl für Programmiersprachen, Erlangen, 1976.
- [Wu 70] WULF, W.A., et alii: "BLISS Reference Manual", Bericht des Computer Science Department, Carnegie-Mellon Universität Pittsburgh, Pennsylvania, 1970.
- [Zi 77] ZISCHLER, H.: Arbeitstitel: "Interaktive Graphendarstellung am Bildschirm", Studienarbeit am Lehrstuhl für Programmiersprachen, Erlangen, erscheint demnächst.

LEBENS LAUF

10. 12. 1943 Geburt in Bamberg. Zweites Kind der Eheleute
Dr. jur. Horst Brüggmann und Frau Charlotte,
geborene Gutgesell
- 1949 bis 1954 Volksschule in Bamberg
- 1954 bis 1963 Humanistisches Gymnasium in Bamberg
- 1963 bis 1964 Studium der Mathematik an der Universität
Erlangen-Nürnberg
- 1964 bis 1966 Freiwilliger Wehrdienst bei der Luftwaffe der
Deutschen Bundeswehr
- 1966 bis 1973 Studium der Mathematik und der Informatik an
der Universität Erlangen-Nürnberg
- seit 1973 Verwalter einer wissenschaftlichen Assistenten-
stelle beim Lehrstuhl für Programmier- und Dia-
logsprachen im Institut für Mathematische Ma-
schinen und Datenverarbeitung der Universität
Erlangen-Nürnberg

Bisher sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen :
=====

Band 1
=====

- NR. 1 Schmitt, A. :
Theorie der nichtdeterministischen und unvollständigen Mealy-Automaten.
(Januar 1967)
- NR. 2 Göring, W., Henning, W., Rasche, W. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 1: Übersicht.
(Juli 1967)
- NR. 3 Namneck, P. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 2: Über Verfahren zur syntaktischen Analyse von
Sätzen formaler Sprachen.
(Juli 1967)
- NR. 4 Eggers, B. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 3: Möglichkeiten der automatischen Erkennung
und selbständigen Korrektur von syntaktischen Fehlern
in formalen Sprachen.
(Januar 1968)
- NR. 5 Schneider, H. J. :
Programmierung numerischer Probleme.
(Oktober 1967)
- NR. 6 Wolf, F. :
Untersuchungen über den Vorhersagefehler bei
stochastischen Automaten.
(Februar 1968)
- NR. 7 Klar, R. :
Rechenautomaten 1.
(April 1968)
- NR. 8 Schmitt, A., Fehling, P. :
Rechenautomaten 2.
(November 1968)
- NR. 9 Weiss, S. :
Untersuchungen zur Kommunikation und Mustererkennung,
Teil 4: Ein Programmsystem zum Testen von Mustererken-
nungsverfahren unter Verwendung eines Sichtgerätes.
(Juli 1968)
- NR. 10 Dokumentation über die Arbeit des Instituts in den
Jahren 1966 bis 1968.
(Januar 1969)

Band 2

=====

- NR. 1 Vollmar, R. :
Über einen Automaten mit Speicherband.
(Januar 1969)
- NR. 2 Händler, W. :
Design Philosophy for Computer Systems.
(März 1969)
- NR. 3 Brackmann, U., Händler, W., Lange, M., Rasche, W.
Wolf, H. P. :
Beiträge zur Kommunikation Mensch - Maschine.
(April 1969)
- NR. 4 Haferkorn, J., Schmitt, A., Seidel, R., Wolf, F. :
Rechnergestützter Unterricht.
(April 1969)
- NR. 5 Böhme, J. F., Göring, W. :
Zwei Beiträge zur Mustererkennung.
(August 1969)
- NR. 6 Köhl, J. :
Datenreduktion zur Speicherung ebener Figuren.
(April 1969)
- NR. 7 Haferkorn, J., Henning, W., Rasche, W. :
Ein bildschirmorientiertes Betriebssystem für die
PDP 7.
(Mai 1969)
- NR. 8 Saffert, J. :
Mikroprogramm-Steuerwerke.
(Juni 1969)
- NR. 9 Spies, P. P. :
Rechnerstrukturen und Ablaufgeschehen in Rechen-
automaten - Analysis und Simulation.
(September 1969)
- NR. 10 Händler, W., Wolf, F. :
Bericht über eine USA-Reise - Aktuelles aus dem
Gebiet der Informatik.
(September 1969)
- NR. 11 Dokumentation über die Arbeit des Instituts im
Jahre 1969.
(Januar 1970)

Band 3

=====

- NR. 1 Schmitt, A. :
Bildschirme und Kommunikation Mensch - Maschine.
(Februar 1970)
- NR. 2 Böhme, J. F. :
Experimente mit stochastischen Automaten.
(Juli 1970)
- NR. 3 Schneider, H. J. :
Chomsky-Systeme für partielle Ordnungen.
(August 1970)
- NR. 4 Haas, G. :
Programme zur dialogischen Logik.
(Oktober 1970)
- NR. 5 Weicker, R. :
Erkennung von kontextfreien Sprachen durch
Tabulator-Turingmaschinen.
(Juli 1970)
- NR. 6 Schmitt, A. :
Optimale Vorhersage der Ausgabe stochastischer
Automaten über lange Zeiten.
(Februar 1971)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1970.
(Februar 1971)

Band 4

=====

- NR. 1 König, R. :
Reduktion und Zerlegung von Automaten.
(Juni 1971)
- NR. 2 Klar, R. :
Messung von Rechneraktivitäten.
(Juli 1971)
- NR. 3 Augustin, U. :
A Note on Arbitrary Varying Channels with
Noiseless Feedback.
A Note on Channels with Finite Memory.
(August 1971)
- NR. 4 Brüggmann, H. W. :
SIDIPRO, Dialogsprache und Echtzeit-Übersetzer
für die Programmierung eines Hochleistungs-
Sichtgerätes.
(August 1971)
- NR. 5 Händler, W. :
Strukturen von Computer-Graphics Systemen.
(Oktober 1971)
- NR. 6 Henning, W. :
Programmierung eines graphischen Sichtgerätes.
(April 1972)
- NR. 7 Dokumentation über die Arbeit des Instituts im
Jahre 1971.
(April 1972)

Band 5

=====

- NR. 1 Steierer, H. F. :
Algorithmen zur Zustandszerlegung von Automaten
unter besonderer Berücksichtigung der graphen-
theoretischen Aspekte.
(März 1972)
- NR. 2 Händler, W., Weizenbaum, J. :
Display Use for Man-Machine Dialog.
(September 1972)
- NR. 3 Dürre, K. :
Erfahrungen mit dem Programmiersystem LEKTOR-1
für Programmierten Unterricht.
(September 1972)
- NR. 4 Vollmar, R. :
SICELA - ein Interpretierer zur Simulation von
Automatennetzen.
(September 1972)
- NR. 5 Spies, P. P. :
Informatik III und Informatik IV.
Vorlesungsskriptum
(Oktober 1972)
- NR. 6 Görz, G. :
Die Sprache GEDANKEN als Beitrag zur syntaktischen
und semantischen Definition von Programmiersprachen.
(Oktober 1972)
- NR. 7 Buchzik, H. :
Maschinen-erzeugte Zufallszahlen.
(Dezember 1972)
- NR. 8 Dokumentation Über die Arbeit des Instituts im
Jahre 1972.
(April 1973)

Band 6

=====

- NR. 1 Domesle, R. :
Vergleich von top-down- und bottom-up-
Kellerautomaten.
(Dezember 1972)
- NR. 2 Brüggmann, H. W. :
Zugriffsstrategien bei Rotationsspeichern.
(Juli 1973)
- NR. 3 Fachgespräch über mehrdimensionale Sprachen.
Erlangen, 14.-15. Juni 1973
- NR. 4 Händler, W. :
Bericht über einen USA-Aufenthalt 1972/1973
Aktuelles aus dem Gebiet der Informatik.
The Concept of Macropipelining.
(Mai 1973)
- NR. 5 Müller, H. :
Diskrete algebraische Strukturen.
(September 1973)
- NR. 6 Müller, H. :
Programmschemata mit Prozeduren.
(Oktober 1973)
- NR. 7 Leeb, K. :
Pascal Theorie.
(Dezember 1973)
- NR. 8 Chatterji, B. N. :
Two-stage Reinforcement Learning System.
An Algorithm for Economical Storage in Dynamic
Programming Approach to Pattern Recognition.
(Dezember 1973)
- NR. 9 Dokumentation über die Arbeit des Instituts im
Jahre 1973.
(Januar 1974)

Band 7

=====

- NR. 1 Eichmeier, P., Goeb, H. :
Anwendungsbeispiele von H-Graphen und VDL
zur semantischen Beschreibung.
(April 1974)
- NR. 2 Händler, W. :
Unconventional Computation by Conventional
Equipment.
(Mai 1974)
- NR. 3 Keramidis, S. :
Ein Beitrag zur Theorie der Prozesssysteme.
(August 1974).
- NR. 4 Nagl, M. :
Formale Sprachen von markierten Graphen.
(August 1974).
- NR. 5 Göttler, H. :
ALGOL 68 - Eine exemplarische Einführung.
(November 1974)
- NR. 6 Altmann, W. :
Konstruktionen zur Erweiterbarkeit von Sprachen.
(November 1974)
- NR. 7 Bunke, H. :
Beschreibung eines syntaxgesteuerten inkrementellen
Compilers durch Graph-Grammatiken.
(Dezember 1974)
- NR. 8 Jahresbericht 1974 der Informatik.
(März 1975)

Band 8

=====

- NR. 1 Ehrig, H., Nagl, M., Schneider, H. J. :
Three Papers on Graph Rewriting Systems and their
Applications.
(Januar 1975)
- NR. 2 Müller, H. :
Diskrete algebraische Strukturen II.
(April 1975).
- NR. 3 Klar, R., Wichmann, H. :
Mikroprogrammierung.
(Juli 1975)
- NR. 4 Weicker, R. :
Turingmaschinen mit assoziativem Speicherzugriff.
(Juli 1975)
- NR. 5 Schneider, H. J. :
Lexikon zur Programmierung I (ALGOL 60).
(Juli 1975)
- NR. 6 Woitok, R. :
Ein neuer Interpreter-Algorithmus für LISP
und seine Korrektheit.
(Juli 1975)
- NR. 7 Luft, A. :
Ein auf der Floyd/Hoare'schen Zusicherungsmethode
aufbauendes Verifikationsverfahren und seine
Begründung.
(August 1975)
- NR. 8 Rohrer, H. :
A Supervised Network of Adaptive Automata for
Pattern Recognition.
Drewes, B. :
Implementation of a Structural Pattern Recognition
System.
(Dezember 1975)
- NR. 9 Klar, R., Schreiber, H. :
Messungen mit dem ZÄHLMONITOR II.
(Dezember 1975)
- NR. 10 Jahresbericht 1975 der Informatik.
(März 1976)

Band 9
=====

- NR. 1 Brendel, W., Frühauf, T., Nagl, M. :
Graph Rewriting Systems II.
Three Papers on Theory, Application, and Implementation.
(Februar 1976)
- NR. 2 Brügmann, H. W., Rudert, A., Lehr, H. :
EBIS
Strukturierte Bildprogrammierung auf höherer Ebene.
(Februar 1976)
- NR. 3 Hofmann, W., Wichmann, H. :
Konzept der Vertikalverarbeitung.
Implementierung und Analyse vertikaler Algorithmen
auf einem mikroprogrammierten Rechner.
(Mai 1976)
- NR. 4 Jordan, H. F., Smith, B. J. :
The Hardware-Software Interface.
(Juni 1976)
- NR. 5 Gernoth, B., Schmidt, B. :
Das Modell MEM-SIM.
Beschreibung eines Simulationsmodells zur Unter-
suchung einer Rechanlage mit einer symmetrischen
Mehrprozessorkonfiguration und privaten Speichern.
(Oktober 1976)
- NR. 6 Grötsch, E. :
Vergleichende Studie über parallele Ersetzung
markierter Graphen.
Jahn, B. :
Eine Studie zur Implementierung von Graph-Grammatiken
unter Verwendung von Assoziativverarbeitung.
(Dezember 1976)
- NR. 7 Mertens, P., Neuwirth, W., Purucker., G.,
Schaller., K., Schmitt., W. :
Verknüpfung von Daten- und Methodenbanken, dargestellt
am Beispiel der Analyse von Marktforschungsdaten.
(Dezember 1976)
- NR. 8 Parallelismus in der Informatik.
Wissenschaftliches Kolloquium aus Anlass des Zehnjährigen
Bestehens des IMMD, Erlangen, 10./11. Juni 1976.
(Dezember 1976)
- NR. 9 Jahresbericht 1976 der Informatik.
(März 1977)

Band 10
=====

- NR. 1 Hoffmann, W. :
Modelle und Analysen von Teilnehmerrechensystemen,
angewendet auf das Teilnehmersystems RESPOND unter
CD 3300 MASTER.
(Februar 1977)
- NR. 2 Körber, K., Werzinger, G. :
Das Modell SIM-LAB.
Simulation eines klinisch-chemischen Laboratoriums.
(März 1977)
- NR. 3 Altmann, W., Weber, D. :
Programmierungsmethodik.
(März 1977)
- NR. 4 Maehle, E. :
Fehlertolerante Rechnerstrukturen.
(April 1977)
- NR. 5 Beth, T. :
Kombinatorik: Theorie der Vertretersysteme.
(April 1977)
- NR. 6 Mertens, P. :
Einige Hinweise zur technischen Gestaltung von
Bildschirmdialogen.
(Mai 1977)
- NR. 7 Jäpel, D. :
Ein Begriffssystem zur Formulierung von Schutzaspekten
und seine Anwendung auf Programmsysteme.
(Juni 1977)
- NR. 8 Brendel, W., Grötsch, E., Nagl, M., Schneider, H.-J. :
Graphersetzungs-systeme, Beiträge zur Theorie, Anwen-
dung und Implementierung.
(Juni 1977)
- NR. 9 Rüdell, W. :
Darstellung und Vergleich abstrakter Modelle zur
Beschreibung geschützter Systeme.
(Juli 1977)
- NR. 10 Bolch, G., Gall, R., Grassi, M., Schmidt, B. :
Modelle für Trommelspeicher.
(Juli 1977)
- NR. 11 v. Eichhorn, K. :
Steiner Systeme, Codes und affine Räume.
(Juli 1977)
- NR. 12 Göttler, H. :
Zweistufige Graphmanipulationssysteme für die
Semantik von Programmiersprachen.
(Juli 1977)

Band 10 (Forts.)

=====

- NR. 13 Keramidis, S. :
 Ein allgemeines formales Modell für Betriebs-
 systeme und dessen Implementierung.
 (September 1977)
- NR. 14 Arnold, F., Benker, L., Grütz, M. :
 Die Fast Fourier Transformation und Möglich-
 keiten ihrer Implementierung auf EGPA.
 (November 1977)
- NR. 15 Gardill, R., Händler, W., Hessling, H.,
 Klar, R., Spies, P. P. :
 ERES
 Eine nichtprozedurale Rechnerentwurfssprache
 mit präziser Zeitbeschreibung.
 (November 1977)
- NR. 16 Wichmann, H. :
 Entwicklung und Bewertung von Algorithmen für
 ein mikroprogrammiertes Parallel- und Assoziativ-
 konzept auf Mikroprogramm- und Benutzerebene.
 (Dezember 1977)
-

Offsetdruck-Fotodruck

J. Hogl, 8520 Erlangen, Hauptstr. 109 (Nähe Martin-Luther-Platz) Tel. 22990